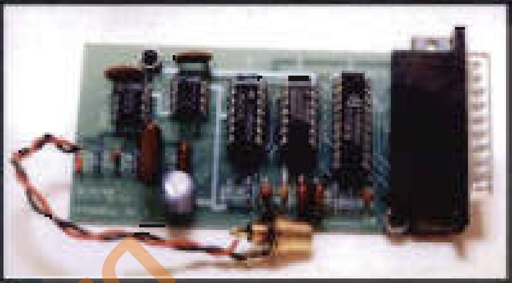
PREMIERE ISSUE

DISK ENCLOSED

AC's TECH For The Commodore AMIGA

Volume 1 Number 1
US $14.95 Canada $19.95

ALL NEW



The VidCell:
Build a 256-Grayscale Digitizer

Interprocess Communication With ARexx

FastBoot:
Developing a Bootable, Recoverable RAM Disk

AmigaDOS For Programmers

Advanced Disassembling:
Magic Macros With ReSource

A Unique Input Device:
Adapting Mattel's Power Glove To The Amiga

And Much More!

ALL TECH

NEW!

Trumpcard Professional...

the Frontrunner in SCSI disk controllers

State of the art users deserve state of the art hardware

Trumpcard Professional is a new generation SCSI controller card for the new generation of 20 Megabit per second drives. While Trumpcard Professional is incredibly fast with drives of any speed, its performance is unmatched with the new generation drives, up to an unprecedented 1.9 MEGABYTES per second with DPERF2! Trumpcard Professional is state of he art. From its surface mounted custom gate array or swift, smooth and seamless data transfers to ts new TCUTILS 2.0 utilities with the look and eel of the new WB 2.0, Trumpcard Professional ets a level of performance and functionality others can only aspire to attain.

Auto mounts all partitions up to 1 minute after RESET, slow partitions don't hold up autoboot.
Full support for removable media built into autoboot ROM. No need for DISKCHANGE.
Full implementation of RDB's and standard Direct SCSI interface.
User selectable Trumpcard SCSI ID for SCSI bus arbitration allows multiple computers to share hard drives.
Supports all drives at 1:1 interleave for fastest data transfers possible.
Full Mac emulator support.
1 year warranty on parts and labor.

laturally, Trumpcard nd Trumpcard 500 wners can pgrade vith IVS' *rumpcard lus $100* pgrade olicy. Call IVS or details. Jpgrade available rom manufacturer only.

Trumpcard 500 Professional is available for A500 users.
NEW!
Lower Price on Standard A2000 Trumpcard!
$169.95

Trumpcard Professional
List Price: $279.95

TRUMPCARD PROFESSIONAL

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

Circle 140 on Reader Service card

AC's TECH/AMIGA

Contents Premiere Issue

6 **Magic Macros with ReSource** *by Jeff Lavin*
Creating image data and reconstructing MFM data are some of the more interesting and useful things that may be done with ReSource.

16 **AmigaDOS, EDIT and Recursive Programming Techniques** *by Mark Pardue*
Developing a hard disk backup utility using only EDIT, AmigaDOS commands, and the magic of recursion.

26 **Building the VidCell 256 Grayscale Digitizer** *by Todd Elliott*
Build an 8-bit video digitizer for less than $80 (including PCB and software).

34 **An Introduction to Interprocess Communication with ARexx** *by Dan Sugalski*
Understanding ARexx's powerful ability to communicate with other programs running simultaneously.

40 **An Introduction to the ilbm.library** *by Jim Fiore*
Speed software development with the ilbm.library's low-level IFF, mid-level IFF, and high-level ILBM calls.

50 **Creating a Database in C, Using dBC III** *by Robert Broughton*
dBC III has applications beyond conventional database applications. It can be used in any situation where referencing data with names is important.

54 **Using Intuition's Proportional Gadgets from FORTRAN 77** *by Joseph R. Pasek*
Using Absoft's FORTRAN 77 to take advantage of most of the Amiga's ROM Kernel without writing extra C or assembly language code.

62 **FastBoot: A Super BootBlock** *by Dan Babcock*
FastBoot is a bootblock that quickly loads an entire disk into memory, creates a RAM disk, and boots from that RAM disk

74 **AmigaDOS for Programmers** *by Bruno Costa*
If you want to delete files, find out file sizes, attributes or the amount of disk space, create or read directories and even run processes from inside your program, read on!

82 **Adapting Mattel's Power Glove to the Amiga** *by Paul King and Mike Cargal*
Construct a special cable and write the necessary software that will interface the Power Glove to the Amiga.

96 **Silent Binary Rhapsodies** *by Robert Tiess*
A poem for programmers

Departments

4 Editorial
49 Source and Executables ON DISK!
81 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
Admin. Assistant:	Alisa Hammond
Circulation Manager:	Doris Gamble
Asst. Circulations:	Traci Desmarais
Corporate Trainer:	Virginia Terry Hicks
Traffic Manager:	Robert Gamble
International Coordinator:	Donna Viveiros
Marketing Manager:	Ernest P. Viveiros Sr.
Marketing Associate:	Greg 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 Editor:	John Rezendes
Video Consultant:	Frank McMahon
Art Director:	William Fries
Photographer:	Paul Michael
Illustrator:	Brian Fox
Graphic Designer:	Kim Kerrigan
Research & Editorial Support:	Alisa Hammond
Production Assistant:	Melissa-Mae Lavoie

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.

IT MAKES A LOT OF SENSE TO BUY A MAST PRODUCT

- **INTERNATIONAL AMIGA PERIPHERAL SUPPLIER** • USA • GERMANY • AUSTRALIA • SWEDEN • UK • AUSTRIA
- **TECHNICAL EXCELLENCE** - **Professionally driven.** 60% of our senior executives are **ENGINEERS**.
- **PERFORMANCE QUALITY PRODUCTS** - We use **PREMIUM COMPONENTS** (FUJITSU Drives for example), **INNOVATIVE DESIGNS** (who else has a SCSI INTERFACE with its own 68000 AND UP TO 2 MEG OF RAM), and some of the most **ELEGANT**, not to mention smallest, external cases that you will see in the industry. Cost effective designs, efficient mass production, and our worldwide resources, bring you high performance products at budget prices. As our President answered to one customer who asked how we do it - **"we're clever and we are not greedy"**

BLITZ BASIC

GIVES YOU CONTROL OVER YOUR AMIGA HARDWARE

CREATE YOUR OWN GAMES, GRAPHICS PRESENTATIONS, AND EDUCATIONAL PROGRAMS WITH MINIMAL PROGRAMMING EXPERIENCE

IT'S NEW, ITS ULTRAFAST, IT'S EASY TO USE

Blitz Basic puts you in control of the Amigas custom sound and graphics chips. **Now you can write sophisticated programs** that previously needed 'C' or Assembler. Blitz is a fully integrated programming language that puts you in control of your Amiga. Unlike s-l-o-w interpreters, Blitz is a true compiler that generates native object code.

Spectacular graphics can be generated with a minimum of commands using the custom chip specific commands included with Blitz. This Basic language enables you to produce QUALITY COMMERCIAL CODE.

$149

FEATURES:

- **Lightning Fast** Compiler • Fast Optimised Object Code
- Rewrite of Amiga Graphics Libraries
- Integrated Editor/Compiler
- Special Effects such as **FADE IN/ FADE OUT**
- Basic Commands to handle IFF Brushes. Anim Brushes and Sound Files
- Direct access to and control of Sprites, Blitter and Audio Hardware
- Supports Dual Playfield, HAM & EHB
- **Queue system** makes blits easy to use
- Number of screens only limited by memory
- Vertical Interupt command allows smooth animation
- Double Buffering, Page Flipping are easily achieved
- **Sound Sequencer** included
- **Machine Language** Subroutines can be added

IMPORTANT NOTE - When you see the demo of **VECTOR BALLS**, remember that the images are being **CALCULATED IN REAL TIME**. This program alone should convince you of the **POWER OF BLITZ**. The source listing will convince you of its **SIMPLICITY**

Demo disks are available for $5. You may also load them off the MAST BBS (702) 359 0132 or (702) 359 0137

FIREBALL - True DMA SCSI interface for the A2000, for sustained performance in a multitasking computer $149

OCTOPLUS - 8 MB RAM for A2000 With 2Mb $269 Extra 2mb $119 GREAT VALUE!.

FANTASTICALLY FANTABULOUS

What else could describe a produduct that amazed visitors to our booth at the LA Ami Expo

$495 COLORBURST $495

TRUE 24 BIT COLOR FOR ANY AMIGA

YES! Every Single pixel on your amiga screen can be any of 16.8 million colors. The color is equal to or better than a Mac. Colorburst supports many video modes and allows digital fade in/out. Dynamic white balance correction is possible. Real time Image processing is a reality. It can be configured as a third hardware playfield allowing Amiga graphics to be overleid - great for animation. We do not use HAM or Composite Y-C mixing, this is pure 24 bit RGB - there is no blurring of colors or loss of definition with colorburst. This product generates BROADCAST QUALITY IMAGES for both PAL & NTSC systems.

FURTHER INFORMATION ON THIS EXCITING PRODUCT IS A
!!!! MUST MAST MUST MAST MUST MAST MUST !!!!
WE CHALLENGE COMPETITORS TO MATCH OUR CLARITY, COLOR & PRICE

STARBLAZER

8 MB RAM & SCSI INTERFACE FOR A500 AND A1000

While many vendors want you to buy a hard drive before memory (because their memory expansions are inside the drive unit), we believe that many customers need memory first. Additionally many people don't want a bulky hard drive right beside the computer, our customers prefer the flexibility offered by an independent external drive that can be placed at a convenient location and even plugged into a different SCSI Interface on another computer. Starblazer is an 8 MB memory and SCSI interface in a minescule case only slightly larger than our very popular Minimegs. It is available populated to 2,4.6.8 megabytes, with or without the SCSI interface installed. **STARBLAZER is real zero wait state ram** that lets your Amiga run at full speed. Dont be misled into buying memory that plugs into the A501 slot and is advertised as "fast ram", it is always slow old chip memory. **Starblazer** includes the MAST **Bytemachine** SCSI Interface. Starblazer Plus includes **WordMASTer** - a SCSI interface that uses a 16 bit interface for high performance. When you need a hard drive, just plug **Tiny** Tiger into the Bytemachine or WordMASTer Interface - it **autoboots, automounts** and is **very fast. Starblazer with 2 MB included from $299. Call for pricing options.**

YOU HAVE TO SEE THIS TO BELIEVE THAT THERE IS SO MUCH POWER IN SUCH A SMALL PACKAGE

Bytemachine $89 WordMASTer $119

ENHANCED UNIDRIVE

The only Amiga external floppy drive in the world that includes **digital track display**, hardware **write protect switch** and inbuilt **hardware virus detection** system - for only $149 why would you want to buy any other. PS. The box looks great.

UNIDRIVE

For those on a budget who still want a **great looking, reliable** and quiet drive, how about the **good ol' Unidrive**. At a list price of **$119**, this is a very attractive deal.

A2000 Internal Floppy $83

Fujitsu Quality Hard Drives

45 meg	$339	90 meg	$539
135 meg	$669	182 meg	$770
410 meg	$1795	672 meg	$2295
1.2 GIG	$3800		

SYQUEST INTERNAL $519
SYQUEST EXTERNAL $625
CARTRIDGES $85

EXTERNAL HARD DRIVES:
Tiny Tiger Deluxe - add $169 to internal drive prices
Tiny Tiger Budget - add $129 to internal drive prices

MAST MIDI INTERFACE with integrated SMPTE to MIDI TIME CODE CONVERTER. $189

DEALERS WELCOME

Send for our color catalogue- there is more than you see here!!

MEMORY AND STORAGE TECHNOLOGY 1395 GREG ST. SPARKS NV 89431 (702) 359 0444
Australia (02) 281 7411 Germany (0221) 771 0918 Sweden (40) 190710 Austria (0)3 16-373763

Circle 180 on Reader Service card.

STARTUP-SEQUENCE

Greetings!

Welcome to the Premiere Issue of *AC's TECH For The Commodore Amiga*. *AC's TECH* is the first all-technical, disk-based magazine designed for Amiga programmers and hardware-types (or wanna-be's). In these pages (and on this disk!) you'll find some the most interesting and informative technical stories ever made available to the Amiga population.

As I said, this magazine was designed for you—the technical Amiga user. You won't find any game reviews in *AC's TECH*. No show reports. Just technical, applications-intensive information.

AC's TECH also includes a disk packed with information, source code, executables, and other technical goodies (we really had no choice—we simply couldn't publish the accompanying multiple 50-page listings!). There is so much information on the disk that we had to archive many of the directories. There's lots to play with!

Whether you build the 256-grayscale digitizer, or explore the possibilities of IPC with ARexx, or work with any of the other superb stories in this issue of *AC's TECH*, you're sure to be *amazed!*

Now, down to business!

To 2.0, or not to 2.0

I've heard some rumbling on the various networks concerning developers' apprehension to design their applications to take advantage of the features of AmigaDOS release 2, their argument being that they don't want to require AmigaDOS release 2 if it *really isn't available* to every Amiga (for the most part, release 2 is seen on Amiga 3000s). This is a valid point, in that most developers in the Amiga community are small, have limited developmental resources, and need to design their products to be compatible with the greatest number of Amigas in use.

However, there is a catch-22 brewing in the distance. Someday, AmigaDOS release 2 will be available for most Amigas. It will, for many consumers, require a hardware upgrade. These consumers are going to need a good reason to upgrade to AmigaDOS release 2—specifically, software that takes advantage of the new standard requesters, the virtual screen capabilities of the new ECS chips, and all the goodies that come with the AmigaDOS release 2 and the new Enhanced Chip Set. While all this is going on, the *other* computer manufacturers are trying to woo the consumer (our Amiga owner) into purchasing their snazzy new units, quickly gaining ground on the Amiga. Hmmm...wasn't release 2 intended to *add* stride to the Amiga? And it's the software that will show off release 2. It's all in the software! Gee...what a concept.

Software comes from developers. Software will make or break the Amiga. Developers have to take the first step. Sure, it will mean investing in both time and money, but if they want the Amiga to advance, they really don't have much of a choice. It's an investment in the future. After all, if the Amiga doesn't expand, neither will their businesses, or this magazine, or...

Stand and deliver

I'm making a big fuss about developing for AmigaDOS release 2; however, you won't find any stories in this issue that cover release 2 specifics. Why? The information in *AC's TECH* is not basic information. *AC's TECH* is not a programmer's reference. Rather, it is a forum for new techniques and innovations for programming and hardware devotees. Right now there really isn't any innovation in the AmigaDOS release 2 arena. We are still learning the basics of programming for AmigaDOS release 2. However, that small learning curve is quickly past.

I am calling upon the readers who have transcended that learning curve. I want to hear from the people who are programming with AmigaDOS release 2. What are you doing? What are the good points and bad points? What about tips and techniques? Are you using GadTools? What about programming with boopsi? We're looking for applications. Let me know!

Send your letters to:

Ernest P. Viveiros, Jr.
AC's TECH/Project Release 3
P.O. Box 869
Fall River, MA 02720-0869

I promise to read every letter. After all, it's the Amiga's future we're dealing with here.

Of course, we are eager to cover any type of true Amiga innovation—be it AmigaDOS 2.0 manipulation, a wicked hardware hack or a new algorithm to generate objects in 3-dimensional space. Our scope is Amiga technical innovation in general, and we intend to cover it precisely and completely.

Ernest P. Viveiros, Jr.
Editor

☑

The Best Assembler

Macro68

Suggested retail price: US$150

Macro68 is a powerful new assembler for the entire line of Amiga personal computers.

Macro68 supports the entire Motorola M68000 Family including the MC68030, MC68882 FPU, and MC68851 MMU. The Amiga Copper is supported also.

This fast, multi-pass assembler uses the new Motorola M68000 Family assembly language syntax, and comes with a utility to convert old-style syntax source code painlessly. The new syntax was developed by Motorola specifically to support the addressing capabilities of the new generation of CPUs.

Macro68 boasts macro power unparalleled in products of this class. There are many new and innovative assembler directives. For instance, a special structure offset directive assures maximum compatibility with the Amiga's interface conventions. A user-accessible file provides the ability to customize directives and run-time messages from the assembler. An AREXX(tm) Interface provides "real-time" communication with the editor of your choice. A number of directives enable **Macro68** to communicate with AmigaDos(tm).

Possibly the most unique feature of **Macro68** is the use of a shared-library, which allows *resident* preassembled include files for incredibly fast assemblies.

Macro68 is compatible with the directives used by most popular assemblers. Output file formats include executable object, linkable object, binary image, and Motorola S records.

Requires at least 1 meg of memory.

ReSource, the powerful disassembler for the Amiga that has received rave [illegible] now has a big brother.

Like the original version, **ReSource'030** will tear apart your code like no other program. And it will do so even faster now, because **ReSource'030** is written in native MC68030 code. This means that it won't run on a vanilla 68000, but will fly on an A3000, or another machine with a 68020/030 board.

ReSource'030 supports the new Motorola M68000 Family assembly language syntax, and is a perfect companion to **Macro68**.

If you're new to **ReSource**, here are a few facts: **ReSource** is an intelligent interactive disassembler for the Amiga programmer. **ReSource** will enable you to explore the Amiga. Find out how your favorite program works. Examine your own compiled code.

ReSource will load/save *any* file, read disk tracks, or disassemble directly from memory. Symbols are created automatically, and virtually *all* Amiga symbol bases are supported. Additionally, you may create your own symbol bases.

"If you're serious about disassembling code, look no further!"

The original **ReSource** continues to be available for owners of 68000 based machines. Both versions of **ReSource** require at least 1 meg of ram.
Suggested retail prices: Original **ReSource**, US$95, **ReSource'030**, US$150

ReSource

The Best Disassembler

PF

The Puzzle Factory, Inc.
P.O. Box 986
Veneta, OR 97487
Orders: (800) 828-9952
Customer Service: (503) 935-3709

Distributors for the U.S. and Canada **Dealer Inquires Invited**

"Quality software tools for the Amiga"

VISA, MasterCard, check or money order accepted - no COD's.
Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc.

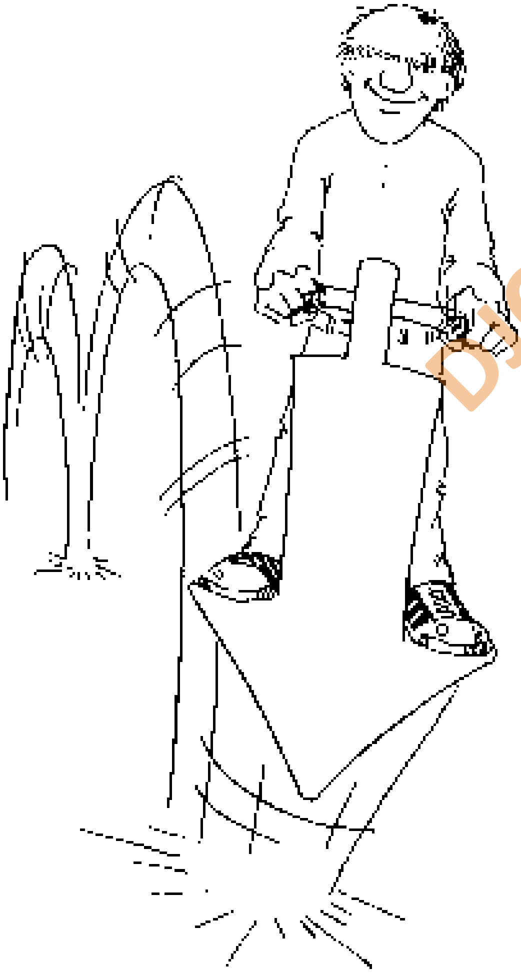
Circle 164 on Reader Service card.

Advanced Disassembling Magic Macros with ReSource

by Jeff Lavin

Introduction

This article is not intended to be a primer on disassembly. We assume that you know how to disassemble programs, and have a need to do so. What I hope to do here is show some of the more interesting and useful things that may be done with ReSource. Some of these examples will use the macro facilities built into ReSource. Macros are a very powerful concept. Just as macros may be used to make your life easier and cut down on typing in a macro assembler, it is possible to think of ReSource as a macro disassembler. Macros similar to the ones shown may be constructed for most repetitive uses. Macros are capable of testing values, and branching based on the outcome.



Fun with Gadgets

Since the Amiga's operating system and library calls depend heavily on structures, it is rare to find a program that does not contain several or perhaps hundreds of structures.

After disassembling a program, you may find the following in a data hunk:

```
ok_Gad      dl      exit_Gad
            dl      $750033
            [illegible] $1A0019
            [illegible] [illegible]
            dw      $100
            dl      OK_Bord
            dw      0
            dw      0
            dl      OK_Text
            dw      0
            dw      0
            dw      0
            dw      0
            dw      0
            dw      0
            dw      1
```

A macro called "Gadget" (see Listing one) is invoked at this point, using "LOCAL MACROS/execute/Gadget". It is important to note several things about this macro:

(1) It works on any (non extended) gadget.
(2) All the data type conversions and other operations are done with one keystroke, invoking the Gadget macro.
(3) When the macro gets finished with one gadget, it automatically moves to the next gadget in the chain. This saves a lot of work when disassembling programs with 50 gadgets. Forward referencing automatically stops when a null pointer is reached.

The result is the very readable gadget structure below.

```
OK_Gad      dc.l     Exit_Gad
            dc.w     157,18,115,13
            dc.w     GADGHCOMP
            dc.w     (RELVERIFY!ENDGADGET)
            dc.w     (REQGADGET!BOOLGADGET)
            dc.l     OK_Border
            dc.l     0
            dc.l     OK_IText
            dc.l     0
            dc.l     0
            dc.w     0
            dc.l     1
```

Creating Image Data

In this example, we have a program that uses various gadgets and images. There is one image that we would like to modify and use in another program. You could use a Clip utility to save the image as a brush, and then use another utility to convert the brush to source code, but it is very easy to get ReSource to do all the conversions in one step.

Here is the data that we found by tracing backwards from gg_GadgetRender -> ig_ImageData:

```
MyImage     dc.l     7FFFFC00
            dl       [illegible]
            dl       [illegible]
            dl       7FFEC0F8
            dl       [illegible]
            dl       [illegible]
            dl       FFFEC000
            dl       [illegible]
            dl       [illegible]
            dl       [illegible]
            dl       [illegible]
            dl       [illegible]
```

The next thing we want is the width of the image. In this case ig_Width is equal to 15 so we set the data type to words this being the lowest multiple of 16 greater than ig_Width. Repeated use of the functions:

```
DISPLAY/Set numeric base/Binary
CURSOR/Relative/Next line * 1
```

gives us these nine arrows to use in our code immediately.

```
MyImage     dc.w     %1111111111111111
            dc.w     %0000000000000000
            dc.w     %0000000000000000
            dc.w     %0000001110000000
            dc.w     %0000011111000000
            dc.w     %0001111111100000
            dc.w     %0001101110110000
            dc.w     %0001101110110000
            dc.w     %0000011111000000
            dc.w     %0000011111000000
            dc.w     %0000011111000000
            dc.w     %0000000000000000
            dc.w     %1111111111111111
            dc.w     %0000000000000000
            dc.w     %0000001111000000
            dc.w     %0000001110000000
            dc.w     %0000001110000000
            dc.w     %0111111111111000
            dc.w     %0001100111000000
            dc.w     %0000111111100000
            dc.w     %0000001110000000
            dc.w     %0000000100000000
            dc.w     %0000000100000100
            dc.w     %0000000000000000
```

If the width is between 17 and 32, we would set the data type to longs. Generally, find out if the width is a multiple of words or longwords, and set the data type appropriately. For really wide images, use your editor to combine lines, using commas as separators.

Effective Address Conversion

Suppose you are disassembling a program. Normally, the first thing you do is to find out if a base register is being used for global variables and data structures. Address register A4 or A5 is generally used for this purpose. If a base register is being used, you tell ReSource where it is pointing; that is, establish the relative-base. ReSource then converts base offsets for you, from the form:

```
move.l   #$48454C50,($26A,A5)
```

to the more readable:

```
move.l   #$48454C50,(lb021844-DT,A5)
```

where the symbol "DT" is the relative base, and is automatically produced by ReSource, if not supplied by you. Please note that nothing has really changed here. The values $26A and lb021844-DT in this program are identical. The obvious difference at this point is that the code is much easier to read.

Even more important is the fact that what was previously defined as a simple offset is now defined as <label> <label>. Down the road, when you attempt to reassemble this file, <lb021844-DT> will be identical to $26A *only* if the data area between both labels later changed; that is, if the labels are still the same distance from each other. Changing the relative position of either label makes the "$26A" offset incorrect, and results in a bad executable, but <lb021844-DT> will assemble to the correct offset for this data item.

Back to our program. When you disassemble this particular program, the first thing you find is:

```
move.l   #(MEMF_PUBLIC!MEMF_CLEAR),d1
move.l   #$56A,d0
movea.l  (AbsExecBase),a6
jsr      (_LVOAllocMem,a6)
movea.l  d0,a5
beq.     [illegible]
[illegible]
```

It quickly becomes obvious in perusing the code that A5 is being used as the base register, but how do you convert all the offsets? It would be so much simpler if an absolute address was being used as the relative-base. This type of situation

What exactly is the ReSource Diassembler?

ReSource is an interactive disassembler for Amiga computers. There are several versions of ReSource available; all require Amiga DOS V1.2 or later and at least 1 meg of RAM. The original ReSource runs on any Amiga and produces conventional 68000 syntax assembly output. ReSource'030 runs only on machines equipped with a 020/030 CPU, and generates Motorola new syntax code. ReSource'068 also produces new syntax code, but will run on any Amiga. All examples herein were produced using ReSource'030.

If you would like more information, please contact:

The Puzzle Factory, Inc.
P.O. Box 986
Veneta, OR 97487
(503) 935-3709

comes up fairly often, not only with allocated memory, but also with programs that use the stack for storage.

In this case, we load the file into ReSource again, but this time as a *binary* file, rather than as an executable, so that ReSource doesn't strip the hunk information. The first thing we need to do is use "DISPLAY/Set data type/Longs" to make the entire file display as longwords. After defining some hunk types, it now looks like this:

```
        dl      HUNK_HEADER
        dl      0
        dl      1               ;Number of hunks in this file
        dl      0
        dl      0
        dl      $3D4            ;Number of longwords of memory
                                ;to allocate for this file
        dl      HUNK_CODE
        dl      $3D4            ;Number of longwords
                                ;to copy into this hunk
        dl      [illegible]
```

What we are going to do is create some extra room at the end of the file for our base-relative variables. We know from the code that the allocated memory is cleared, so we don't have to worry about static data structures being referenced. We also know that the program is asking for $548 bytes of memory. This is equal to $152 longwords, so we need to change the value at offset $14 to ($3D4 + $152) or $526 longwords. If the size of the allocated memory is not evenly divisible by 4, just round up.

Select "SPECIAL FUNCTIONS/Zap" and when the requester comes up, we enter "L$526". The "L" tells ReSource to change a longword rather than a word or byte. Then select "SAVE/Save binary image/all" and save the file under a *different name*.

Now load the modified file, this time as an executable. When we move to the end of the file we see:

```
        dx.b    $548        ;the size of our original work area.
```

which is exactly what we wanted! After placing this line at the cursor position, select:

```
SPECIAL FUNCTIONS/Specify base register/A4     ;Change base
                                               register
SPECIAL FUNCTIONS/Convert (xx,A4) EA's/This address
```

and at the top of the file (remember, there was only 1 hunk!), now see:

```
DT          equ     [illegible]    ;offset from start of program to
                                   ;work area.
```

This is our new relative-base. Memory is still allocated and (we fervently hope!) freed, but when this file is reassembled, the "DX" (Define eXtra) area at the end of the program will hold our variables. Selecting "PROJECT/Disassemble" indeed shows that all our base-relative offsets have been converted. We now have a much easier time figuring out how this program works.

Most of the time this hack works perfectly. After all, Lattice and Manx have been doing it for years! Sometimes, however, you will convert a program and it won't run. Although the AmigaDOS file system is beyond the scope of this article, there are several "gotchas". The main thing to remember is not to add "DXs" to an empty hunk. If you find an empty hunk at the end of the file, just before the "DXs", either add a dummy line of data, or delete that SECTION statement, and add the "DXs" to the end of the previous hunk.

Here are a few thoughts on "DXs". This 'data type' results in much smaller load files. The size of the file on disk does not include the "DXs". The main difference between a "DX" and a BSS hunk is that the "DX" must be cleared by your program, not by the loader. This is easy enough to accomplish with the following code, which preserves D0/A0:

```
            lea     DT,a4               ;Our relative base
            lea     (BSS_start-DT),a1   ;Start of the "DXs"
            move.w  #(BSS_len/4)-1,d2   ;Size of the "DXs" in bytes
            moveq   #0,d1
1$          move.l  d1,(a1)+
            dbra    d2,1$
```

Although they have been available to Lattice and Manx C users for years, "DXs" have only been available to assembly language programmers for a short time, and only on a few assemblers. DigiSoft's Macro68 is one assembler that supports this feature.

Coping with Library Stubs

Anyone who has ever attempted to disassemble a program written in C, in the era before pragmas and inline code, has come up against library stubs. A library stub is a horrible little piece of code that takes up space and converts C arguments (passed on the stack), into system arguments (passed in registers). Here's one now:

```
lb013932   movea.l  (lb01F12C-DT,a4),a6
           movea.l  (4,sp),a1
           move.l   (8,sp),d0
           jmp      (-$228,a6)
```

There are frequently anywhere from 50 to hundreds of these stubs in a program; some of them are not even referenced! ReSource makes it very easy to convert the library offset to human readable form. After you find out what's in the library base register (A6), select the appropriate library symbol base:

```
lbC013932  movea.l  (_SysBase-DT,a4),a6
           movea.l  (4,sp),a1
           move.l   (8,sp),d0
           jmp      (_LVOOpenLibrary,a6)
```

But there remains that unreadable label! It is easy to type labels by hand when there are just a few, but after pages of them, it gets old fast! The macro "Create LibCall Names" (see Listing two) solves this problem easily.

Basically, the macro starts by searching for either of these strings:

```
           jsr      (_LVO
           jmp      (_LVO
```

When one is found—for example, "jsr (_LVOOpenLibrary"—ReSource gets the symbol, which in this case is "_LVOOpenLibrary", and puts it in the accumulator, which is the main string-manipulating buffer in ReSource. The "_LVO" at the beginning is clipped off, and a "_" is added to the front. Now we have "_OpenLibrary", which works nicely as a label. ReSource then moves back to the previous label, "lbC013932", and creates a new label for us. Now, our library stub looks like this:

```
_OpenLibrary movea.l  (_SysBase-DT,a4),a6
             movea.l  (4,sp),a1
             move.l   (8,sp),d0
             jmp      (_LVOOpenLibrary,a6)
```

ReSource then searches for the next label, and the macro ends. We can process hundreds of these nasty library stubs quickly, simply by using the "Repeat last command" key—typically, bound to the spacebar.

Reconstructing MFM Data

Losing data through failure of magnetic media must surely be one of the most frustrating experiences. Although many people use hard drives now, most still need floppy disks for backup purposes, and when your hard drive "goes down", the last thing you need to find out is that your backup disks have a read error.

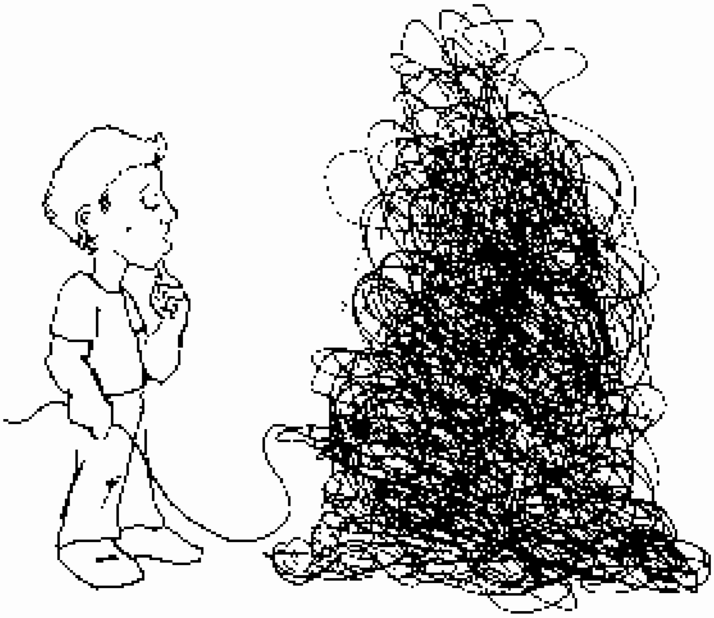
Many floppy disk read errors are actually the result of only *one bit* being wrong. As a complete disk contains over seven million good bits, it follows that if the bad bit can be identified and fixed, most or all of the data can be recovered. Using the philosophy that the problem is really only in a specific area on the disk, we will attempt to analyze that area, and then to recover as much data as we can.

First, use the "PROJECT/Read tracks" function in ReSource to find the unreadable track. Next, we need a separate program capable of reading raw MFM data, not the system data that sector editors read. It must also be capable of reporting the address of its track buffer. There are available several disk utilities, or monitors, that meet these criteria.

Use that utility to read the raw track. While the raw data is still in the utility's buffer, switch to ReSource, and select "PROJECT/Disassemble memory". When the requester comes up, enter the address of the disk utility's track buffer, and use the "SAVE/Save binary image/All" function to make this into a normal binary file. After loading this file into ReSource by using "PROJECT/Open binary file", you may exit the utility, and start examining the raw track data in ReSource.

What we will be attempting to do next is to recover as many good sectors from the track as possible, by converting each sector from MFM to hex individually. While at the start of the file, select the "DISPLAY/Set data type/Words" function. To find the start of the first sector, look for the hex word "$4489". Scroll to just past this word. If the next word is also "$4489", scroll past it also. Set the data type to longwords, using "DISPLAY/Set data type/Longwords". Use the "Convert MFM encoding" macro, and if the sector header is okay, it should now have some full line comments on the current line:

```
; Format type = $FF
; Track number = 2
; Sector number = 7
; Sectors to EOT = 4
                 d0          $552AAA2A
                 d1          $552AA529
                 d1          $4AAAA4A4
                 d2          $2AA55AAA
```



The format byte should always be $FF, whenever reading Amiga disks. The track number should be the same as the one that you originally read. The sector number must be 0-10 inclusive, and "Sectors to GAP" must be equal to eleven minus the sector number. If any of these conditions fail, chances are that either you haven't found the true start of a sector, or the sector header information itself is damaged.

The "Calculate MFM xsums" macro works out the header checksum, and the data block checksum, as they are defined in the header block itself:

```
; Format byte = $FF
; Track number = 0
; Sector number = 0
; Sectors to GAP = 2
                dl      $552AA524
                dl      $552AAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
; Header block checksum = $[illegible]
                dl      $AAAAAAAA
                dl      $AAAAA523
; Data block checksum = $10115541
                dl      $AAAAAAAA
                dl      $92A955A9
                dl      $2AAAAAA4
```

The full-line comment (above) giving the data block checksum is followed by the two longwords making up the checksum. Immediately following this is 1024 bytes of MFM data. Scroll to the start of the MFM data, and use the "Conv MFM sector to HEX" macro to convert each longword of MFM into hex. To convert a complete sector, this must be done 128 times. The macro "Convert MFM 128 times" will do this. Note that the macro line "GLOBAL MACROS/Execute/#14" refers to the macro number of "Conv MFM sector to HEX", and may be different on your system. After converting to hex, the section start looks something like this.

```
; Format byte = $FF
; Track number = 0
; Sector number = 10
; Sectors to GAP = 2
                dl      $552AA529
                dl      $5122AAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
                dl      $AAAAAAAA
; Header block checksum = $00010500
                dl      $AAAAAAAA
                dl      $AAAAA529
; Data block checksum = $10011161
                dl      $2AAAAAAA
                dl      $[illegible]
```

```
                dl      $2AAA9AA4               ; $0000[illegible]
                dl      $55551952               ; $0000[illegible]
                dl      $94AAAAAA               ; $0000[illegible]
                dl      $2AAAAA54               ; $0000[illegible]
                dl      $55512229               ; $0000[illegible]
                dl      $9[illegible]           ; $1857D32E
                dl      $12[illegible]          ; $3522[illegible]
                dl      $94[illegible]          ; $0[illegible]
                dl      $AA52D252               ; $[illegible]
                dl      $51122111               ; $17[illegible]
                dl      $9[illegible]           ; $[illegible]
```

Depending on what is actually wrong with the track, you will normally be able to salvage 10 of the eleven sectors, and possibly some of the bad sector as well. Salvaging data is very difficult to completely automate, but when you really need to salvage as much data as possible, these macros may make the job much easier for you. The hex data is displayed in the end-of-line comments above.

Because there may be many things wrong with the data, and our space is limited, we will not discuss various methods of repairing sector headers, disk tracks, or rewriting the reconstructed data out to disk. There are programs available to do this sort of thing with less work and less opportunity for human error. Our purpose here is to illustrate, in a general way, how the macro capabilities of ReSource may be used for complex tasks.

That's All Folks!

Because it is a *macro* disassembler, the things that may be done with ReSource are limited only by your imagination. I hope I have shown a representative sample of some of the things it is possible to do with ReSource.

About the Author

Jeff Lavin and his wife Grace own *The Puzzle Factory*, which publishes the ReSource disassembler, and Macro68 assembler. Jeff was introduced to assembly language during the homebrew-computing days on his SYM-1, and has been programming in assembly ever since. You may contact Jeff through The Puzzle Factory, or write to him c/o AC's TECH.

The following macros may be entered into ReSource, after which you may want to save them, by selecting "LOCAL MACROS/Save all macros".

Listing One
The GADGET macro

```
LABELS/Edit single/Full-line comment                  ;Create a blank line above gadget.
DISPLAY/Set data type/Longs                           ;Pt. to next gadget.
CURSOR/Relative/Next line * 1                         ;Move to next line
DISPLAY/Set data type/Words                           ;These are all WORDS.
CURSOR/Relative/Next line * 4                         ;Move past next 4 words and change
DISPLAY/Set data type/Bytes                           ;data type
CURSOR/Relative/Previous line * 4                     ;Move back to coordinates.
DISPLAY 2/Multiple constants override/Set             ;Force coordinates all on 1 line
DISPLAY/Set Numeric base/Decimal                      ;and display them in decimal
CURSOR/Relative/Next line * 1                         ;Move to next line.
DISPLAY/Set data type/Words                           ;These are also WORDS.
SYMBOLS 2/E-G/Gadget flags                            ;Equate gadget flags.
CURSOR/Relative/Next line * 1                         ;Move to next line.
SYMBOLS 2/E-G/Gadget activation                       ;Equate gadget activation
CURSOR/Relative/Next line * 1                         ;Move to next line.
SYMBOLS 2/E-G/Gadget types                            ;Equate gadget types.
CURSOR/Relative/Next line * 1                         ;Move to next line
DISPLAY/Set data type/Longs                           ;These are all LONGS.
CURSOR/Relative/Next line * 5                         ;Skip down 5 lines.
DISPLAY/Set data type/Words                           ;Set gg_GadgetID to WORD.
DISPLAY/Set Numeric base/Decimal                      ;and display it in decimal.
CURSOR/Relative/Next line * 1                         ;Move to next line.
DISPLAY/Set data type/Longs                           ;Set gg_UserData to LONG.
CURSOR/Relative/Next line * 1                         ;Move to next line.
LABELS/Edit single/Full-line comment                  ;Create a blank line.
CURSOR/Relative/Previous line * [illegible]           ;Move back to top of gadget.
CURSOR/Absolute/Forward reference                     ;Follow reference to gg_NextGadget.
```

Listing Two
The CREATE LIBCALL NAMES macro

```
CURSOR/Normal search/Set search string "j[mp|sr]<tab>_LVO"
CURSOR/Pattern search/Find next occurrence           ;Either "jsr _LVO" or "jmp _LVO".
STRINGS/Get/Symbol                                   ;Get symbol in the accumulator.
STRINGS/Edit functions/Clip start "_LVO"             ;Clip out the "_LVO".
STRINGS/Edit functions/Prepend "_"                   ;Add "_" to the front.
CURSOR/Relative/Previous label                       ;Find this stub's label.
STRINGS/Put label                                    ;Create our new label
CURSOR/Relative/Next label                           ;Halt at next possible stub.
```

Listing Three
Convert MFM encoding macro

```
STRINGS/Accumulator/Hex                              ;Ensure accumulator displays HEX
STRINGS/Default size/Longword                        ;Ensure default size = longword
STRINGS/Get/Cursor longword                          ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"      ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional              ;Don't abort macro if above
                                                     ;returned zero

CURSOR/Relative/Previous line - 1                    ;Move up one line
CURSOR/Copy/Clip #1                                  ;Remember current cursor position
                                                     ;within file

CURSOR/Relative/Next line + 1                        ;Go to next line
STRINGS/Maths functions/Add $1D,$1D                  ;Whatever is in accumulator, add
                                                     ;to itself!

LOCAL MACROS/Directives/End conditional              ;Don't abort macro if above
                                                     ;returned zero

STRINGS/Swap with buffer.../"A"                      ;Store result in buffer "A"
CURSOR/Relative/Next line + 1                        ;Go to next line
STRINGS/Get/Cursor longword                          ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"      ;AND out unwanted bits
STRINGS/Maths functions/Logical OR $1D,"A"           ;Logically OR with contents of
                                                     ;buffer "A"

CURSOR/Relative/Previous line - 1                    ;Go back one line
STRINGS/Define string/"A" $1D,$1D                    ;Copy accumulator to buffer "A"
STRINGS/Edit functions/Clip end "??????"             ;Clip last six characters from
                                                     ;accumulator

STRINGS/Edit functions/Prepend "Format byte = "      ;Add required text
LABELS/Edit single/Full-line comment $1D,$1D         ;Create comment, using accumulator
STRINGS/Define string/Acc $1D,"A"                    ;Copy buffer "A" to accumulator
STRINGS/Edit functions/Clip start "??????"           ;Strip unwanted characters
STRINGS/Edit functions/Clip end "????"               ;Strip unwanted characters
STRINGS/Edit functions/Prepend "$"                   ;Put dollar sign back
STRINGS/Accumulator/Decimal                          ;Require number displayed in
                                                     ;decimal

STRINGS/Maths functions/Increment                    ;Add 1 to accumulator
LOCAL MACROS/Directives/End conditional              ;Don't abort if above returned
                                                     ;zero

STRINGS/Maths functions/Decrement                    ;Subtract 1 (required number is
                                                     ;decimal)

LOCAL MACROS/Directives/End conditional              ;Don't abort if above returned
                                                     ;zero

STRINGS/Edit functions/Prepend "Track number = "     ;Add required text
LABELS/Edit single/Full-line comment $1D,$1D         ;Create comment, using accumulator
STRINGS/Define string/Acc $1D,"A"                    ;Copy buffer "A" to accumulator
STRINGS/Edit functions/Clip start "?????"            ;We want 2nd add this time
STRINGS/Edit functions/Clip end "??"                 ;Don't want 3rd
STRINGS/Edit functions/Prepend "$"                   ;Put dollar sign back
STRINGS/Maths functions/Increment                    ;Convert hex to decimal
LOCAL MACROS/Directives/End conditional              ;Convert hex to decimal
STRINGS/Maths functions/Decrement                    ;Convert hex to decimal
LOCAL MACROS/Directives/End conditional              ;Convert hex to decimal
STRINGS/Edit functions/Prepend "Sector number = "    ;Add required text
LABELS/Edit single/Full-line comment $1D,$1D         ;Create comment, using accumulator
STRINGS/Define string/Acc $1D,"A"                    ;Copy buffer "A" to accumulator
STRINGS/Edit functions/Clip start "????????"         ;Require LSB this time
STRINGS/Edit functions/Prepend "$"                   ;Put dollar sign back
STRINGS/Maths functions/Increment                    ;Convert hex to decimal
LOCAL MACROS/Directives/End conditional              ;Convert hex to decimal
STRINGS/Maths functions/Decrement                    ;Convert hex to decimal
LOCAL MACROS/Directives/End conditional              ;Convert hex to decimal
STRINGS/Edit functions/Prepend "Sectors to GAP = "   ;Add required text
LABELS/Edit single/Full-line comment $1D,$1D         ;Create comment, using accumulator
CURSOR/Paste/Clip #1                                 ;Go back to original line -1
CURSOR/Relative/Next line + 1                        ;Finish on original line
```

Listing Four
Calculate MFM
xsums macro

```
CURSOR/Relative/Next line + 10                          ;Move down 10 lines
STRINGS/Set radix/Hex                                   ;Set calculator to hex
STRINGS/Operand size/Longword                           ;Set accumulator size to longword
STRINGS/Get/Cursor longword                             ;Get a longword into accumulator
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional                 ;Don't abort if above returned zero
STRINGS/Swap with buffer, / A                           ;Store result in buffer "A"
CURSOR/Relative/Next line + 1                           ;Move to next line
STRINGS/Get/Cursor longword                             ;Get a longword into accumulator
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Swap with buffer, / A                           ;Store result in buffer "A"
STRINGS/Maths functions/Add $15,$17                     ;Whatever is in accumulator, add to
                                                        ;itself!
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Maths functions/Logical OR $13,"A"              ;Logically OR with contents of
                                                        ;buffer "A"
STRINGS/Edit functions/Prepend "Header checksum = "     ;Add detailed text
CURSOR/Relative/Previous line + 1                       ;Move up one line
LABELS/Edit single/Full-line comment $13,$15            ;Create comment, using accumulator
CURSOR/Relative/Next line + 2                           ;Move down two lines
STRINGS/Get/Cursor longword                             ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Maths functions/Add $13,$15                     ;Whatever is in accumulator, add to
                                                        ;itself.
LOCAL MACROS/Directives/End conditional.                ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Swap with buffer, / A                           ;Store result in buffer "A"
CURSOR/Relative/Next line + 1                           ;Move down one line
STRINGS/Get/Cursor longword                             ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
STRINGS/Maths functions/Logical OR $13,"A"              ;Logically OR with contents of
                                                        ;buffer "A"
CURSOR/Relative/Previous line + 1                       ;Move up one line
STRINGS/Edit functions/Prepend "Data block checksum = " ;Add detailed text
LABELS/Edit single/Full-line comment $13,$15            ;Create comment, using accumulator
CURSOR/Relative/Previous line + 8                       ;Move down 8 lines
```

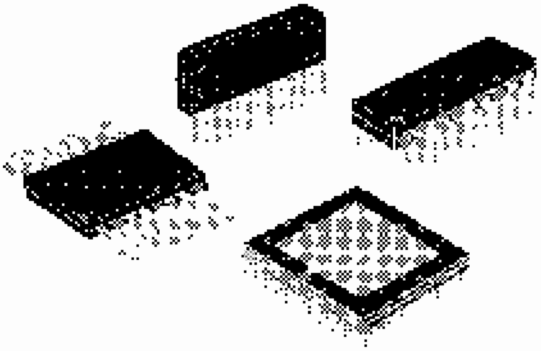
Listing Five
Convert MFM sector to
HEX macro

```
CURSOR/Copy/Clip #2                                     ;Remember this position!
STRINGS/Get/Cursor offset                               ;Where are we?
STRINGS/Maths functions/Add "$200"                      ;Add sector size to current offset
CURSOR/Absolute/Specify offset $13,$15                  ;Go there!
STRINGS/Get/Cursor longword                             ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Swap with buffer, / A                           ;Store result in buffer "A"
CURSOR/Paste/Clip #2                                    ;Go back to original position
STRINGS/Get/Cursor longword                             ;Get longword at cursor position
STRINGS/Maths functions/Logical AND "$55555555"         ;AND out unwanted bits
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Maths functions/Add $15,$17                     ;Whatever is in accumulator, add to
                                                        ;itself.
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
STRINGS/Maths functions/Logical OR $13,"A"              ;Logically OR with contents of
                                                        ;buffer "A"
LOCAL MACROS/Directives/End conditional                 ;Don't abort macro if above returned
                                                        ;zero
LABELS/Create single/End-of-line comment $16,$12        ;Create end-of-line comment
CURSOR/Relative/Next line + 1                           ;Move down one line
```

Listing Six
Convert MFM 128 times macro

```
STRINGS/Define string/ C "128"                ;Number of times to repeat
LOCAL MACROS/Set macro label/*1               ;Mark this position in macro
GLOBAL MACROS/Execute/ (#14)                  ;Macro function to be repeated
STRINGS/Swap with buffer../ C                 ;Swap accumulator with buffer "C"
STRINGS/Math functions/Decrement              ;Subtract 1 (assumes number is decimal)
STRINGS/Swap with buffer../ C                 ;Swap accumulator with buffer "C"
LOCAL MACROS/Previous macro label/*1          ;Repeat until zero
```

✓



AC/BASIC

HIGH PERFORMANCE

- Up to 50X increase in execution speed over interpreted BASIC
- Fully compatible with Amiga BASIC interpreter
- Compiler generates fast, native 68000 machine code
- Support for 16/32 bit integers and 32/64 bit floating point numbers
- Supports both IEEE and BCD (Decimal) math
- Program and data sizes limited only by available memory and disk capacity
- Fast compiler; processes thousands of lines per minute

PROFESSIONAL FEATURES

- AC/BASIC is a complete package. Unlike other BASIC compilers, no additional libraries are required for ROM kernel access, and no "runtime modules" are required to distribute compiled applications
- Complete 100% access to ROM kernel O/S services
- Absolutely NO fees or charges to distribute compiled programs
- Batch compiling
- Environmental editor allows customizing runtime environment
- Dynamic runtime linking, or link runtime during compile

EASY TO UNDERSTAND AND USE

- Full BASIC reference manual included. Every BASIC statement is completely defined and explained
- Compiler runs from Workbench or CLI and has intuitive user interface
- Working examples provided in manual and on disk
- Large example programs included. Great for learning or for use in your own software

FULLY COMPATIBLE WITH INTERPRETER

- Develop applications with interpreter, then turn up the speed by compiling into stand-alone applications, with no source code changes
- Compiler implements Microsoft BASIC language definition for maximum compatibility with interpreter

ENHANCING OLDER PROGRAMS

- Multi-way logic made simple with `SELECT CASE` statement
- `STATIC` keyword for extremely fast array processing
- Dynamic subprograms support recursion

CREATE EASILY PORTABLE APPLICATIONS

- AC/BASIC runs on Amiga, Apple IIGS, and Macintosh (as Microsoft BASIC compiler)
- Standard language definition simplifies machine ports
- Port applications to/from Macintosh, IBM PC, and MS DOS environments

Also Available — AC/FORTRAN

VAX/VMS compatible ANSI 77 compiler with full screen source level symbolic debugger, linker, library manager, IEEE math, C interface, supports virtual arrays,. No limit on code or data size. 68020/030 version also available.

absoft

High Performance Scientific/Engineering Software

2781 Bond Street
Rochester Hills, MI 48309 USA
Tel: (313) 853-0050
Fax: (313) 853-0108
BBS: (313) 853-0000

ALL BRAND OR PRODUCT NAMES ARE TRADEMARKS OF THEIR RESPECTIVE HOLDERS

Circle 184 on Reader Service card.

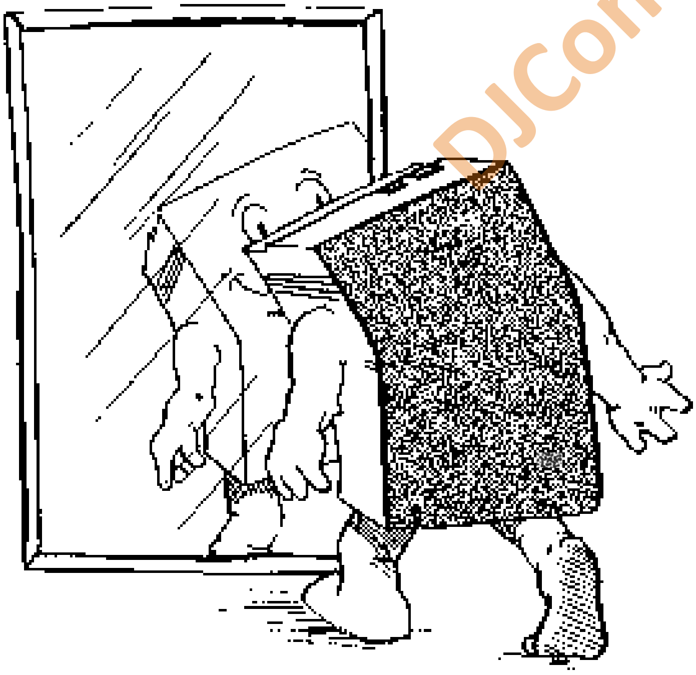
The Use of Recursive Programming Techniques in Conjunction With DOS and EDIT for Hard Disk Backup

by Mark D. Pardue, PhD

The objectives of this article are:

(1) To demonstrate the use of recursive programming techniques;
(2) To also demonstrate the use of files as "templates" for EDIT commands; and
(3) To present an example hard disk utility developed using only DOS commands and the EDIT line editor.

This utility was developed for a particular Amiga configuration, and as such, is useful only as a demonstration of the concepts discussed here, unless it is customized to work with other configurations.



Introduction

The development of this hard disk backup utility came about one day as I attempted to justify keeping the line editor EDIT on my hard disk, in addition to Ed and TxEd, which are my workhorse editors. Further, I had been trying for several months to find a utility that would allow me to back up my primary hard disk (DH0:) to my secondary hard disk (JH0:)—an IBM hard disk connected through the A2088 Bridgeboard on my Amiga 2000. Now, I realize that I could copy all files from DH0: to JH0: every time that I wanted to do an incremental back up, but this is an unduly lengthy process. I could also buy a C compiler (or other language software) and write a backup program, but I didn't want to spend the money. Finally, I could use good old AmigaBASIC, but I just don't like it that much (so much for user preferences).

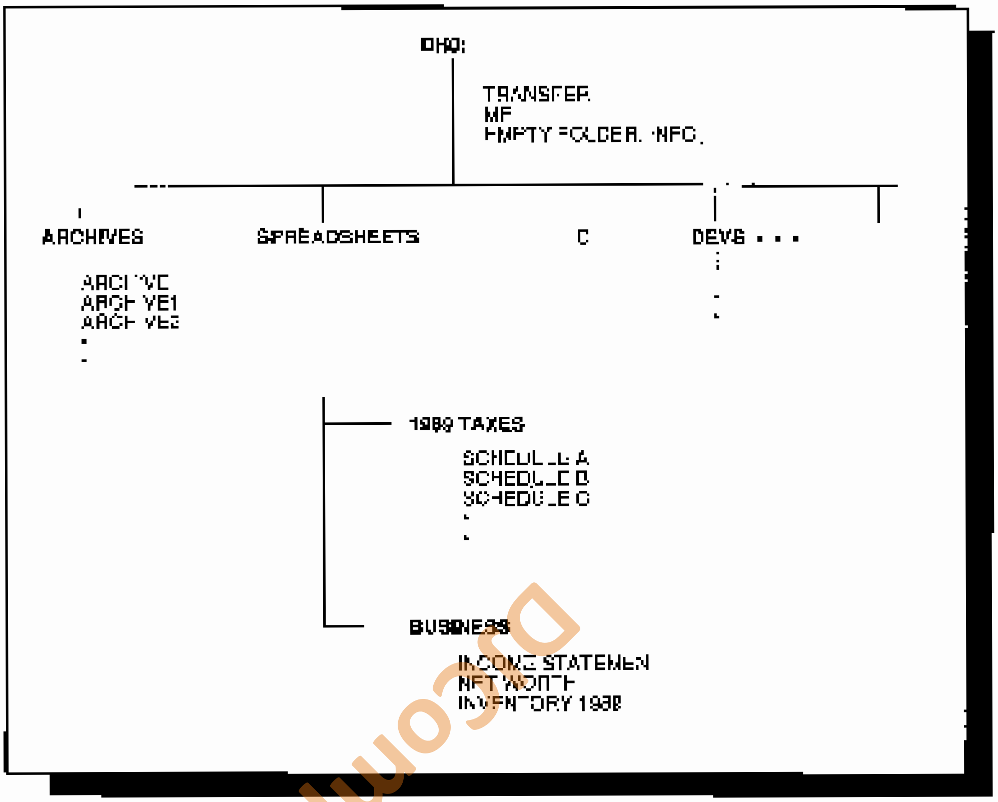
So, I decided to investigate developing the hard disk backup utility using DOS, thinking that would provide an easy, straightforward solution. I'm not sure how easy it was, but by doing it, I learned a lot of useful details about DOS, EDIT, and recursive programming techniques. The purpose of this article is to pass some of that information along to other Amiga users, and hopefully provide a simple hard disk backup utility that people can use until such time as they can afford a good commercial backup program.

Hard Disk Backup Using DOS Functions

As you know, AmigaDOS has no built-in backup function such as that found in MS-DOS. There are several commercial

Figure 1

Example Directory/Subdirectory Structure



backup programs available, but I wanted to build one myself, using only DOS functions. I did not want the files compressed; in fact, I wanted the same file and directory structure on the backup disk so that I could immediately operate directly from that disk if my primary disk failed. The COPY and PROTECT commands can be used to first copy a file and then set the "archive" bit for that file. However, there is no direct way to have the COPY command copy only those files that do not have their archive bit set (i.e., those files that have changed since they were last archived).

AmigaDOS does provide a way to get a list of files and the status of their archive bits through the LIST command. The LFORMAT option of the LIST command even lets the user tailor the output to a degree. However, there is no method to have the LIST command list only those files with archive bit not set.

One solution to this problem is to use the LIST command to redirect the output to a file, then to edit that file with the line editor EDIT to develop a list of those files with archive bit not set. Finally, this list of files can be further edited to insert "COPY" before each filename to create a script file for copying those files that have changed since the last backup. Executing this script file copies the changed files, and then the files only need to have their archive bit set to indicate that they have now been backed up. Unfortunately, there is no easy way to accomplish this (again unlike MS-DOS, which allows the copy and archive bit setting to take place together, using the XCOPY command). However, we can repeat a part of our process above, substituting "PROTECT +a " for the word "COPY" in the last file edit.

A Special Feature of Edit, the Line Editor

At this point in the development process, a problem arose: "How do I edit the file listings output from the LIST command?" The same editing commands have to be used every time the backup utility program is run, but I definitely wanted the editing to be automatic. I'm sure that there are other ways to automate such a process using ARexx, C, or Modula-2, but not having any of these tools (like many Amiga owners, I'm sure), I turned to EDIT.

As mentioned above, one day I was doing housekeeping on my hard disk, trying my hardest to find a reason to keep EDIT on the disk. I use Ed and Uedit regularly, and one more line editor seemed unnecessary. But I kept EDIT on my hard disk, due mainly to its unique capability to edit one file using another file of EDIT commands. Later, I used it in developing this utility.

Recursion

The basic program structure was now established. The LIST command could be used to list just the files in a directory, using the FILES option, and then these files could be backed up. The LIST command could then be repeated to list only subdirectories in this particular directory, using the DIRS option. At this point, the entire process of listing files, backing them up, then listing the subdirectories in the directory needed to be repeated for each of the subdirectories in the first (sub)directory encountered (see Figure 1). This process would go on and on until the bottom of the directory tree was reached. The program must return to the top of the tree and then repeat the process.

over and over, until all directories have been backed up. This type of program lends itself to a technique known as recursion.

Recursion is the process whereby, in the execution of a program, that program calls itself repeatedly until execution is complete. In the case of a backup utility, the program need only back up one directory, then go into each of its subdirectories and call itself. When it calls itself in the first subdirectory, that subdirectory becomes the directory that the newly-called program acts upon. The original calling program waits until the called program has completed its backup of the first subdirectory before it proceeds to backing up the second subdirectory, by calling itself again. The power of recursion, though, lies in the fact that as part of the process of backing up the first subdirectory, the newly-called program may have to call itself to back up a subdirectory of that subdirectory. This nesting of calling and called programs is limited only by the capabilities of the machine that the program is running on, and such things as stack size.

This process of recursion makes our job of developing a backup utility that much easier. We only need to develop a simple program that will back up one directory and then call itself to back up its subdirectories. That program must:

(1) list the files that are to be archived;
(2) copy those files;
(3) set the archive bit of the files;
(4) list the subdirectories, and
(5) go into each subdirectory and call itself.

That is the entire process. The recursion ensures that all files in all subdirectories are backed up, no matter how complicated the directory structure. The process is started, and each piece of the process stops when it is completed. The very first calling program automatically stops executing when all called programs have completed execution. The calling and called programs are simply different invocations of the same program. All of these invocations are kept track of by the operating system.

That's all there is to recursion. It has an almost magical way of performing a complicated task with a very simple program. It just breaks down a complicated task into smaller, less complicated tasks, each of which is accomplished by the same simple program.

Details of the Hard Disk Backup Utility

The backup utility is started by a script file appropriately called BACKUP. The listing for BACKUP is shown in Figure 2. It first clears the screen, then uses PROTECT to set the archive bit for any file that is not to be backed up (in my case, an MS-DOS backup file). Next, it displays a message that the backup is beginning. Then, the recursive program ARCHIVE is called. This is the program that actually performs the backup; it will be discussed in detail below. It resides in the ARCHIVES directory along with all of the support files required for its operation. When the ARCHIVE program has finished execution (i.e., the backup is complete), that message is displayed, temporary working files are deleted, and the directory is changed to the desired current directory (in my case, DH0:).

The listing for the ARCHIVE program is shown in Figure 3. Each line of the program will be explained separately below.

```
(1)    .KEY path,slash
```

As you can see, the parameters <path> and <slash> are passed to the program via .KEY. The <path> and <slash> parameters are appended to the phrases "DH" and "JH", and are passed down through the recursive calling of the program. BACKUP passes the initial values of <path> as 0 and slash as :. Later, these parameters are used to give the phrases "DH0:" and "JH0:", the desired source drive and destination drive, respectively. This somewhat unorthodox setting of these parameters is required to support the rest of the program calls. For example, on later calls of the program, <path> could be set to 0:Spreadsheets and <slash> could be set to /. This allows the filename TAXES to be appended to the phrases to give "DH0:Spreadsheets/TAXES" and "JH0:Spreadsheets/TAXES" for source filename and destination filename, respectively. This will be demonstrated as more of the program details are explained below.

```
(2)    .DEF slash /
```

Except for the first invocation of the program when <slash> is set to :, all other occurrences of <slash> are desired to be the slash symbol used to set off directories. By declaring the default value of <slash> to be / we do not need to pass it to future calls to ARCHIVE.

FIGURE 2
STARTUP FILE FOR HARD DISK BACKUP UTILITY

```
CLS
PROTECT "DH0:DOS/MS-DOS BACKUP" -a       ;don't backup this file
ECHO " "
ECHO "    BEGINNING BACKUP"
ECHO " "
EXECUTE DH0:ARCHIVES/ARCHIVE 0 :         ;0 is <path> : is <slash>
ECHO " "
ECHO "    BACKUP COMPLETE"               ;all done
ECHO " "
DEL RAM:ARCHLIST                         ;clean up temporary files
DEL RAM:ARCHDIRY
DEL RAM:ARCH.WORK??
DEL RAM:WORK
CD DH0:                                  ;set default directory
```

FIGURE 3
ARCHIVE PROGRAM FOR HARD DISK BACKUP UTILITY

```
.KEY path,slash                                            ;path and slash &
                                                           ;slash is / or /

.DEF slash /                                               ;Default is "/"

FAILAT 30                                                  ;set returncode limit
                                                           ;enough to allow for
                                                           ;edit and copy errors
                                                           ;[illegible]

CD "DH0:<path><slash>"                                     ;advance to new path


IF <slash> EQ /

    IF NOT EXISTS "<path>"                                 ;[illegible] directory [illegible]
        ECHO "ERROR: DIRECTORY [illegible] <path>"
        [illegible] "[illegible]"
    ENDIF

ENDIF


LIST > RAM:ARCHLIST NOHEAD FILES                           ;list files

EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE1 TO RAM:TEMP         ;[illegible] files
EDIT > NIL: RAM:TEMP     WITH DH0:ARCHIVES/ARCHIVE2 TO RAM:ARCHLIST     ;[illegible]

IF NOT WARN                                                ;[illegible]

    EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE3 TO RAM:TEMP     ;copy script
    EDIT > NIL: RAM:TEMP     WITH DH0:ARCHIVES/ARCHIVE4 TO RAM:ARCHCOPY ;add "path"

    ECHO "COPYING FILES FROM DH0:<path><slash>"
    EXECUTE RAM:ARCHCOPY "<path>" <slash>                  ;execute copy script


    EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE5 TO RAM:ARCHPROTECT   ;protect script

    EXECUTE RAM:ARCHPROTECT                                ;execute protect script

ENDIF
        ;directories


LIST > RAM:ARCHLIST NOHEAD DIRS                            ;list dirs
EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE6 TO RAM:TEMP         ;add [illegible]


IF ERROR                                                   ;no dirs?
    SKIP END                                               ;[illegible]
ENDIF


EDIT > NIL: RAM:TEMP     WITH DH0:ARCHIVES/ARCHIVE7 TO RAM:ARCHLIST     ;add "path" [illegible]
EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE8 TO ARCHDIRS         ;[illegible] path


EXECUTE "RAM:ARCHDIRS" "<path><slash>[illegible]"          ;execute ARCHIVE for
                                                           ;each directory

CD "DH0:<path><slash>[illegible]"                          ;[illegible]

LAB END
```

(3) FAILAT 30

The returncode must be set to at least 30 to allow execution of the program to continue if an error occurs in the EDIT commands or the PROTECT. This can occur when EDIT tries to find a match in the file listing (described below) and one is not found, which in this program just means that all files in a particular directory have not changed since the last backup. And, when used for some older software such as Graphicraft, the PROTECT fails and an error is generated. These programs never have their archive bit set, and subsequently are backed up every time you execute the backup program. This causes no harm, and in both of these cases it is desireable to have the program continue executing because these errors have no effect on the proper operation of the program.

FIGURE 4
SUPPORT FILES FOR HARD DISK BACKUP UTILITY

ARCHIVE1

```
[illegible]
```

ARCHIVE2

```
[illegible]
```

ARCHIVE3

```
[illegible] <path><slash>"/[illegible]
```

ARCHIVE4

```
[illegible]
```

ARCHIVE5

```
[illegible]
```

ARCHIVE6

```
[illegible] ARCHIVEDIR [illegible]
```

ARCHIVEDIR

```
[illegible]
[illegible]
```

ARCHIVEKEY

```
[illegible]
```

(4) CD "DH<path><slash>"

This command changes execution to the new directory that is to be backed up. The quote marks around the phrase allow filenames and directory names with a space to be used (unlike MS DOS conventions). As discussed above, <path> continues adding more levels of the path as each instance of ARCHIVE is called. For example:

```
Initial call:     CD "DH0:"
2nd call:         CD "DH0:Spreadsheets/"
3rd call:         CD "DH0:Spreadsheets/1988
                      Budgets/"
```

This concept applies throughout the program execution wherever <path> and <slash> occur.

(5) IF <slash> EQ /

The next piece of code creates the target directory, if it doesn't exist. However, we don't want to do that during the very first call to ARCHIVE where <slash> is equal to : and we are backing up files in DH0:. Therefore, we use the IF statement to execute the next piece of code only if <slash> is equal to / (not equal to :).

(6) IF NOT EXISTS "JH<path>"

We want to create the target directory if it does not exist.

(7) ECHO "MAKING DIRECTORY JH<path>"

Lets the user know that the target directory is being made.

(8) MD "JH<path>"

Makes the target directory.

(9) ENDIF

End of the piece of code that makes the new target directory if it doesn't exist.

(10) ENDIF

Ends the piece of code below the check for <slash> equal to /.

(11) LIST > RAM:ARCHLIST NOHEAD FILES

All files in the current directory are listed. The file listing is accomplished through the LIST command using only the NOHEAD and FILES option. The NOHEAD option deletes the header information normally found in the listing using the LIST command, and the FILES option lists only files, no directories. The output of the LIST command is redirected into a file called ARCHLIST (in RAM: for speed).

FIGURE 5
EXAMPLE OUTPUT LISTING FROM STEP (11)

```
TRANSFER                       [illegible] ----rwed Saturday  10:28:07
Clock.info                       634 ----rwed 24-Nov-89 12:16:35
Simulation                      2332 ----rwed 17-Oct-89 20:04:21
Simulation.info                  770 ---arwed 17-Oct-89 20:05:38
Backup                           369 ---arwed Monday    19:13:07
XP                              3588 ----rwed 30-Sep-89 15:20:45
BF-XX                             26 ---arwed 16-Oct-89 19:[illegible]
[illegible]                    50125 ----rwed [illegible]-Mar-89 [illegible]:44:[illegible]
[illegible].info                1156 ----rwed Saturday  [illegible]:26:40
Expansion.info                   [illegible]46 ----rwed [illegible]-Sep-89 13:14:47
Disk.info                        [illegible]92 ----rwed [illegible]-Nov-89 [illegible]:10:34
.info                            272 ----rwed Today     17:58:41
Utilities.info                   494 ---arwed 30-Sep-89 13:03:42
System.info                      494 ---arwed 30-Sep-89 13:03:43
Spreadsheets.info                494 ---arwed 30-Sep-89 13:03:58
Word Processing.info             494 ---arwed [illegible]-89 10:56:19
Preferences.info                 [illegible] ---arwed 30-Sep-89 13:12:39
Empty Folder.info               2546 ----rwed 17-Oct-89 20:05:13
[illegible].info                1210 ----rwed 04-Oct-89 15:08:51
```

(12) EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE1 TO RAM:TEMP

This is where the first EDIT command is executed. The redirection to NIL: keeps all EDIT messages from appearing on the screen (reduces screen clutter and unnecessary information). The filelist in ARCHLIST is edited using the command stored in ARCHIVE1, shown in Figure 4. An example ARCHLIST is shown in Figure 5. We want to save any line in the filelist that has the protection bit "a" missing. We will key on the protection bits field and assume that all files that we want to back up are readable so that the protection bit "r" is set. Therefore, any filelist line with "-r" in the protection bit field designates a file we want to back up, and all others we will assume are either already backed up ("ar") or are unreadable ("a-"). There is one glitch in this, in that a -r in the filename itself will also be backed up, whether desired or not. This is a small price to pay for the simplicity, and for most users this will probably not be a problem.

```
The command "(DF/-r/;N)" in ARCHIVE1:

(        starts at the top of the file
DF       deletes every line in ARCHLIST until
/-r/     the string /-r/ is found, leaves that line
;        ends the DF delete command
N        moves down to the next line
)        repeats the DF delete and next line sequence.
```

Notice the symbol / is used as the delimiter in the EDIT commands. This continues until the end of the file is reached. The resulting edited filelist is shown in Figure 6 and contains the list of files that we need to back up, along with some information that we don't care about. This edited filelist is stored in RAM: (for speed) in a file called TEMP. EDIT does not allow you to edit a file in this manner and just save the newly edited file under the same name, thus, the new filename TEMP.

FIGURE 6
OUTPUT LISTING FROM STEP (12)

```
TRANSFER                       63 ----rwed Saturday  10:23:07
XP                           3588 ----rwed 30-Sep-89 15:20:45
.info                         272 ----rwed Today     [illegible]
Empty Folder.info            2[illegible]6 ----rwed 17-Oct-89 20:05:[illegible]
```

(13) EDIT > NIL: RAM:TEMP WITH DH0:ARCHIVES/ARCHIVE2 TO RAM:ARCHLIST

This command is very similar to the one above. It edits the edited filelist generated above by using ARCHIVE2 (Figure 4). The objective of this EDIT command is to clean up the "garbage" left in our filelist to result in a simple list of files.

```
The command "0;(SB/  /;D;)" in ARCHIVE2:

0        starts at the top of the file
(SB      splits every line when it reaches
/  /     two spaces in the line
;        ends the SB split line command
D        deletes the next line (i.e. the "garbage")
;        ends the D delete line command
)        repeats the SB and D sequence.
```

This continues until the end of the file is reached. The resulting edited filelist is shown in Figure 7 and contains just the list of files that we need to back up. This edited filelist is stored back in RAM:ARCHLIST.

FIGURE 7
OUTPUT LISTING FROM STEP (13)

```
TRANSFER
XP
.info
Empty Folder.info
```

(14) IF NOT WARN

This statement causes the program to skip the next piece of code, which copies and sets the archive bits of the files to be backed up if a WARN returncode is returned by the previous command. If the first edited filelist contains no files (i.e., none of the files in the directory have changed since the last backup), RAM:TEMP will have no entries. When the previous EDIT command is called, an empty file TEMP will cause a WARN returncode.

(15) EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE3 TO RAM:TEMP

This command is similar to the others above. It edits the edited filelist generated above by using ARCHIVE3 (Figure 4). The objective of this EDIT command is to add the COPY command in front of each filename in our filelist to come up with a script file to perform the actual backup.

```
The command "B/E//COPY "DH<path><slash>/;A L// TO
"JH<path><slash>"/;N" in ARCHIVE3:

B           starts at the top of the file
E/          search line and insert before the string
//          nothing (the beginning of the line)
COPY        inserts the COPY command and the phrase
"DH<path>   source pathname
<slash>     the separation slash (or "/")
/           end of phrase to insert at this point
;           ends the E search and insert before command
A           search line and insert after
L           begin search from the end of the line
//          nothing (the end of the line)
TO          inserts the TO keyword and the phrase
"JH<path>   destination pathname
<slash>     the separation slash (or "/")
/           end of phrase to insert at this point
;           ends the A search and insert after command
N           moves down to the next line
;           repeats the E and A sequence
```

This continues until the end of the file is reached. The resulting edited COPY script file is shown in Figure 8. It contains just the list of COPY commands to copy the files that we need to back up from the source DH<path><slash> and to the destination JH<path><slash>, as we desire. This edited COPY script file is stored in RAM:ARCHTEMP because we still have to add .KEY with <path> and <slash> to the script file for it to work properly.

FIGURE 8
OUTPUT LISTING FROM STEP (15)

```
COPY "DH<path><slash>TRANSFER" TO "JH<path><slash>"
COPY "DH<path><slash>KEY" TO "JH<path><slash>"
COPY "DH<path><slash>.info" TO "JH<path><slash>"
COPY "DH<path><slash>Empty Folder.info" TO
                                  "JH<path><slash>"
```

(16) EDIT > NIL: RAM:TEMP WITH DH0:ARCHIVES/ARCHIVE4 TO RAM:ARCHCOPY

This command takes care of the last item identified above—it adds .KEY with the required parameters. It also edits the initial script file generated above by using ARCHIVE4 (Figure 4).

```
The command "I/DH0:ARCHIVES/ARCHIVEKEY/" in ARCHIVE4:

          inserts at the top of the file
/DH0: ... the contents of file ARCHIVEKEY
```

The contents of the ARCHIVEKEY file is .KEY and the parameters required for the script file, <path> and <slash>. The W command in ARCHIVE4 completes the editing, and is required when the I command uses a file for the inserted text. The resulting complete COPY script file is shown in Figure 9 and is stored in RAM:ARCHCOPY.

FIGURE 9
OUTPUT LISTING FROM STEP (16)

```
.KEY path,slash
COPY "DH<path><slash>TRANSFER" TO "JH<path><slash>"
COPY "DH<path><slash>KEY" TO "JH<path><slash>"
COPY "DH<path><slash>.info" TO "JH<path><slash>"
COPY "DH<path><slash>Empty Folder.info" TO "JH<path><slash>"
```

(17) ECHO "COPYING FROM DH<path><slash>"

This command lets the user know which directory is being backed up. With a little more work, you could just as easily print out the name of each program being copied, but to reduce screen clutter I chose to only display the directory.

(18) EXECUTE RAM:ARCHCOPY "<path>" <slash>

This command executes the script file that we developed above to do the copying for this particular directory. Notice that the path variable (the complete pathname for this directory) is passed to the script file with quotes to allow for spaces, and the slash variable is also passed.

(19) EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE5 TO RAM:ARCHPROTECT

This command edits the edited filelist generated several steps above by using ARCHIVE5 (Figure 4). The objective of this EDIT command is to add the PROTECT command in front of each filename in our filelist to come up with a script file to set the archive bit for each of the files that we backed up.

```
   The command [illegible] in
   ARCHIVE2:

T             starts at the top of the file
IB            search line and insert before the string
//            nothing (the beginning of the line)
PROTECT       inserts the PROTECT command and the
"             beginning quote to allow spaces in filenames
/             end of phrase to insert at this point
;             ends the IB search and insert before command
A             search line and insert after
-             begin search from the left until that
//            nothing (the end of the line)
"             inserts ending quote for filenames and a space
+A            and the protection bit to set the archive bit
/             end of phrase to insert at this point
;             ends the A search and insert after command
N             moves down to the next line
;             repeats the T and A sequence.
```

This continues until the end of the file is reached. The resulting edited PROTECT script file is shown in Figure 10. It contains just the list of PROTECT commands to set the archive bit for each of the files that we backed up. This edited COPY script file is stored in RAM:ARCHPROTECT.

FIGURE 10
OUTPUT LISTING FROM STEP (19)

```
 PROTECT "ARCHIVES" +a
PROTECT "XXX" +a
PROTECT "Tools" +a
PROTECT "Empty Folder.info" +a
```

(20) EXECUTE RAM:ARCHPROTECT

This command executes the script file that we developed above to set the archive bit of the backed up files using the PROTECT command. Notice that we do not have to pass either the path variable (the complete pathname for this directory) or the slash variable to the script file. That is because we are setting the archive bit for files in our current directory, rather than performing actions across drives as above.

(21) END

This ends the section of the program that backs up the files in our current directory. Next, we will go into each of the subdirectories in the current directory and back them up separately.

(22) LIST > RAM:ARCHLIST NOHEAD DIRS

All subdirectories in the current directory are listed. The subdirectory listing is accomplished through the LIST command using only the NOHEAD and DIRS option. The NOHEAD option deletes the header information normally found in the listing using the LIST command, and the DIRS option lists only directories, no files. The output of the LIST command is redirected into RAM:ARCHLIST. An example directory list is shown in Figure 11.

FIGURE 11
OUTPUT LISTING FROM STEP (22)

```
ARCHIVES                 Dir    ----rwed Monday     18:54:43
System                   Dir    ----rwed 04-Jan-89  19:44:41
Utilities                Dir    ----rwed Monday     20:53:22
Spreadsheets             Dir    ----rwed Monday     19:55:02
Word Processing          Dir    ----rwed Monday     19:22:12
?                        Dir    ----rwed Monday     20:34:23
Empty Folder             Dir   ----rwed 13-Jan-89  19:15:00
Tools                    Dir    ----rwed Monday     19:15:21
XXX                      Dir   ----rwed 27-May-89  21:15:4[illegible]
Empty                    Dir   ----rwed 30-Sep-89  10:[illegible]
Graphics                 Dir   ----rwed 30-Sep-[illegible] [illegible]
C                        Dir   ----rwed Monday     20:51:[illegible]
L                        Dir   ----rwed 30-Sep-89  16:21:41
Demo                     Dir    ----rwed Monday     19:19:20
S                        Dir   ----rwed Monday     20:34:55
Temp                     Dir   ----rwed 15-Nov-89  23:[illegible]
Expansion                Dir   ----rwed 17-May-[illegible] 2[illegible]:01:[illegible]
Accessories              Dir   ----rwed 31-Dec-89  12:25:31
```

(23) EDIT > NIL: RAM:ARCHLIST WITH DH0:ARCHIVES/ARCHIVE2 TO RAM:TEMP

This is where the first EDIT command is executed for the list of subdirectories in the current directory. The command edits the directory list generated above by using ARCHIVE2 (Figure 4). This step is the same as step (13) above and simply deletes the "garbage" at the end of each line in the listing, as shown in Figure 12.

FIGURE 12
OUTPUT LISTING FROM STEP (23)

```
ARCHIVES
System
Utilities
Spreadsheets
Word Processing
?
Empty Folder
Tools
XXX
Empty
Graphics
C
L
Demo
S
Temp
Expansion
Accessories
```

(24) IF ERROR

This statement causes the program to skip to the end of the program if an ERROR returncode is returned by the previous command. If the edited directory list contains no directories (i.e., there are no subdirectories in the current directory), RAM:ARCHLIST will have no entries. When the previous EDIT command is called, an empty file ARCHLIST will cause an ERROR returncode.

FIGURE 13
OUTPUT LISTING FROM STEP (27)

```
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>ARCHIVES"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>System"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Utilities"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Spreadsheets"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Word Processing"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>T"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Empty Folder"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Fonts"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>ARP"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Empty"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Trashcan"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>C"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>L"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Devs"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>S"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Libs"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Expansion"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Accessories"
```

(25) SKIP END

This statement skips program execution to the END label at the end of the program — we are done backing up the current directory. All files have been backed up and there are no subdirectories to back up.

(26) ENDIF

This is the end of the IF statement for checking on subdirectories.

(27) EDIT > NIL: RAM:TEMP WITH DH0:ARCHIVES/ARCHIVE6 TO RAM:ARCHLIST

This command edits the edited directory list generated above by using ARCHIVE5 (Figure 4). The objective of this EDIT command is to add a command in front of each directory name in our directory list to develop a script file to execute this very same ARCHIVE program we are describing. This is the recursive nature of the program discussed previously.

```
The command "B''EXECUTE DH0:ARCHIVES/ARCHIVEDIR
"<path><slash>'";A ""'"';N!" in ARCHIVE6:

T          starts at the top of the file
B'         search line and insert before the string
""         nothing (the beginning of the line)
EXECUTE    inserts the EXECUTE command and the
DH0:...    filename to be executed
"          beginning quote to allow spaces in the name
<path>     pathname of the current directory
<slash>    / for separating directory and filename
'          end delimiter for insert phrase
;          ends the B search and insert before command
A          search line and insert after
"          begin search line, use null until find
""         nothing (the end of the line)
"          inserts the ending quote for the filename
'          end of phrase to insert at this point
;          ends the A search and insert after command
N          moves down to the next line
!          repeats the B and A sequence.
```

Notice that the delimiters we use with this EDIT command have changed from / to ' to allow / characters to be used in the inserted phrase. This edit continues on each line of the file until the end of the file is reached. The resulting edited directory list is shown in Figure 13. It contains the list of EXECUTE commands to execute a script file named ARCHIVEDIR (shown in Figure 4) for each subdirectory in the current directory. This edited EXECUTE script file is stored in RAM:ARCHTEMP because we still have to add .KEY with <path> and <slash> to the script file for it to work properly.

FIGURE 14
OUTPUT LISTING FROM STEP (28)

```
.KEY path,slash
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>ARCHIVES"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>System"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Utilities"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Spreadsheets"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Word Processing"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>T"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Empty Folder"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Fonts"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>ARP"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Empty"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Trashcan"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>C"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>L"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Devs"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>S"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Libs"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Expansion"
EXECUTE DH0:ARCHIVES/ARCHIVEDIR "<path><slash>Accessories"
```

```
(28)  EDIT > NIL: RAM:ARCLIST WITH
      DH0:ARCHIVES/ARCHIVE4 TO ARCHDIRS
```

This command takes care of the last item identified above – it adds .KEY with the required parameters, and is the same as step (16) above. The resulting complete EXECUTE script file is shown in Figure 14 and is stored in ARCHDIRS. Notice that this file is not stored in RAM:, but in the current directory. Thus, as we go down through the subdirectories, making an ARCHDIRS for each one of them, we can keep them separate until we complete the backup.

```
(29)  EXECUTE "DH<path><slash>ARCHDIRS"
      "<path>" <slash>
```

This command executes the script file that we developed above to back up each subdirectory in this particular directory. Notice that the path variable (the complete pathname for this directory) is passed to the script file with quotes to allow for spaces, and the slash variable is also passed. Looking at the file ARCHDIRS that we created (Figure 14), notice that for each directory, we execute another script file named ARCHIVEDIR (see Figure 4) in the ARCHIVES directory. While this seems like an unnecessarily complex nesting, it is required to keep the <path>, <slash> and <dir> parameters separate. Each EXECUTE command in the developed script file/directory list ARCHDIRS passes a parameter to ARCHIVES/ARCHIVEDIR that serves as <dir>. That parameter consists of the original <path> and <slash> parameters appended to the subdirectory name with quotes around the entire parameter to allow for spaces in the pathname.

For example, suppose we are backing up a directory named "Spreadsheets" which has a subdirectory named "1989 Taxes". The <path> parameter is "DH0:Spreadsheets" the <slash> parameter is /. The parameter passed as <dir> would be "DH0:Spreadsheets/1989 Taxes". Now, this parameter is used as the new <path> parameter for the newly-called instance of ARCHIVES/ARCHIVE done by ARCHIVES/ARCHIVEDIR. The new <slash> parameter is specified by default in line (2) of ARCHIVE to be /, which is what we want. The process starts all over for this new directory, which is the technique of recursion.

```
(30)  DEL "DH<path><slash>ARCHDIRS"
```

When the whole EXECUTE script in ARCHDIRS of the current directory is completed, all of the lower-level subdirectories in the current directory have been completely backed up. Therefore, we no longer need the working file ARCHDIRS stored in our current directory, and can delete it with this command.

```
(31)  LAB END
```

This completes the backup of the current directory – all files in the directory were backed up in step (18) and each of the subdirectories were backed up in step (29), through the use of recursive calls to this same program.

Conclusion

This program is rather simple in concept, as it:

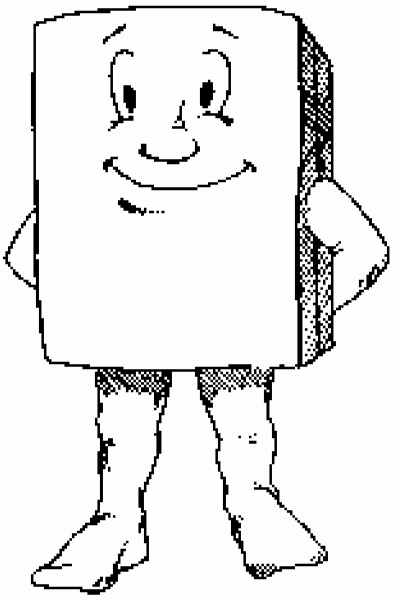
(1) backs up all of the files in a directory,
(2) calls the program again for each subdirectory to backup its own files,
(3) calls the program again for each of its subdirectories to backup those subdirectory's files,
(4) calls the program again, etc

You get the picture: it's kind of like placing two mirrors facing each other and looking at the never-ending reflections. The difference in this case is that when the last subdirectory is backed up, the last instance of the program ends, ending the next-to-last instance of the program, and so on, until the first instance of the program that started the whole process ends.

☑

About the Author

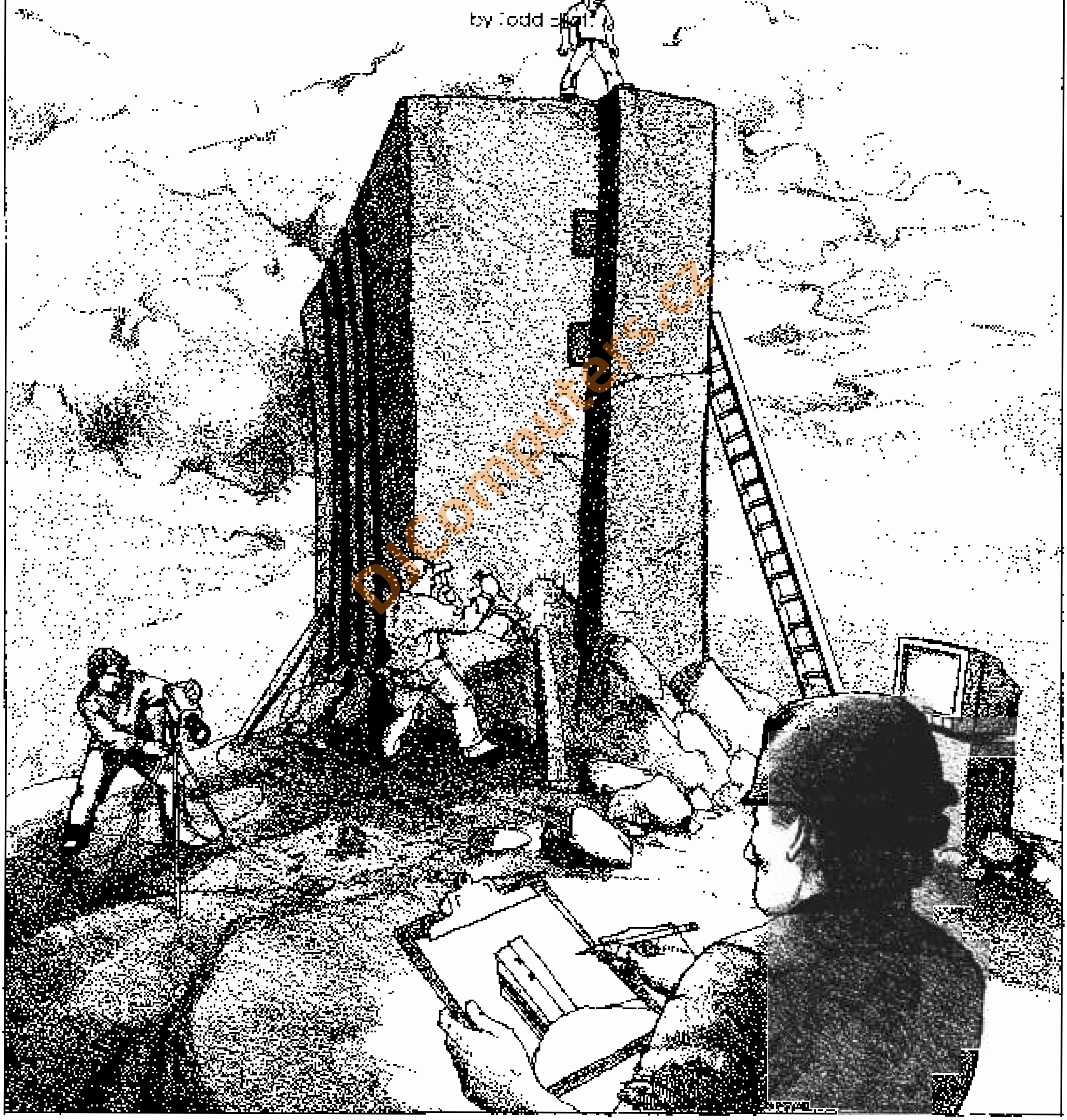
Dr. Pardue is presently an Assistant Professor of Computer Engineering at Old Dominion University, at Norfolk, Virginia. He has been using an Amiga since he bought an Amiga 1000 in 1985, and now uses an Amiga 2000 equipped with a 40 MB Amiga hard disk and an A2088 Bridgeboard with a 32 MB IBM hard disk. You can contact Dr. Pardue c/o AC's TECH.



BUILDING THE VIDCELL 256 GRAYSCALE DIGITIZER

(INCLUDES ALL SOFTWARE!)

by Todd Elliott



Building the Vidcell & Using the Vidcell Software

BUILDING THE VIDCELL

OVERVIEW

One of the best things that people appreciate about the Amiga is its great video capabilities. The new trend in computer video is the digitizing of 24-bit color (8 bits each for red, green, and blue) images. Digitizing is useful for a variety of things, such as placing pictures in your reports, creating a database of images, or placing pictures in your programs.

Digitizing is usually accomplished either by using a color camera, or a black-and-white camera with color filters. This gives the user images of more than 16 million (2 to the 24th power) colors. The most popular video digitizer presently available for the Amiga is a 7-bit (21-bit color) digitizer; it is only capable of displaying 2.1 million colors, however. Color scanners can be used to achieve the new 24-bit images, but these expensive peripherals do not fit into everyone's budget.

This article will show you how to build an 8-bit video digitizer for less than $80 (with a professional circuit board and software), capable of twice as many grayscales as the most popular commercial digitizer now available for the Amiga. If you can solder or know someone who does, you should be able to complete this project without any difficulty.

FROM CAMERA TO COMPUTER

A camera converts an image's light intensity into an analog signal. Before an image can be used by a computer, the analog signal must itself be converted into digital values that represent the analog intensity. This process is known as A/D conversion. In this case, the video digitizer performs this function. The analog signal from a camera is fed into the video digitizer for conversion. The converted values are then fed to the Amiga computer via the parallel port. Once the video data is in the computer's memory, it can then be manipulated in many different ways, and also displayed. The information can then be saved as an IFF picture file for future use with any of the image software packages.

THEORY OF OPERATION

Please refer to the schematic accompanying this article. The video signal from the RS-170 video source (camera, VCR, etc.) is fed to the input of U1. U1 filters out the horizontal and vertical sync signals from the video source. These signals are used to synchronize both the circuitry and the computer to the incoming video information. The computer waits for a vertical sync via pin 10 of the parallel port. Once the signal is received, we know that the camera is at the top of the picture and it is time to start converting the information. Video information is too fast for the computer to keep up with through the parallel port, so this digitizer uses a left-to-right, slow-scan process. The information is collected in vertical columns from left to right at a rate of 1 column every 1/60th of a second. Since there are 640 columns of video on the Amiga, it takes 640 times 1/60th of a second, or 10.6 seconds, to digitize a complete image.

The timing for the left-to-right scan is controlled by the charging of C4. The voltage across C4 is initially brought to zero volts before digitizing is started. This grounding action is accomplished by using the analog switch U4a, controlled by pin 12 of the parallel port. As C4 is allowed to charge, its voltage determines the delay value of the one shot U2. The horizontal sync signals from U1 are fed to the trigger input of the one shot U2. This trigger signal occurs once at the beginning of each horizontal line. The output of the one-shot U2 is delayed by the voltage across C4 and fed to the sample-and-hold section of the circuit (U3, U4, and C5). C5 of the sample-and-hold circuit now holds the analog value of one individual pixel. This analog value is then filtered and fed to the input of the A/D converter U5. U5 converts this value to a digital binary number, which is then made available to the computer via pins 2 through 9 of the parallel port. This process is continued until all 640 vertical columns are collected. Once all the information is in the computer's memory, the binary values for each pixel may be manipulated and displayed. For example, if you want to increase the brightness of the picture, you can add a constant value to each pixel value and display it.

BUILDING THE UNIT

The circuit can be built using the wire-wrapping technique, but I strongly recommend using a printed circuit board, for several reasons. First and most important, the unit works better with a printed circuit board due to RF noise reduction. It is also much easier to put it together without making mistakes, since the circuit board has silk-screened labels for all the parts. Using a printed circuit board also makes the unit much more compact and professional looking. The parts may be ordered from DIGI-KEY, or I can supply the parts with the circuit board (see the complete parts list accompanying this article). I recommend using high-quality parts and sockets for the circuit. This will maximize the performance of your digitizer.

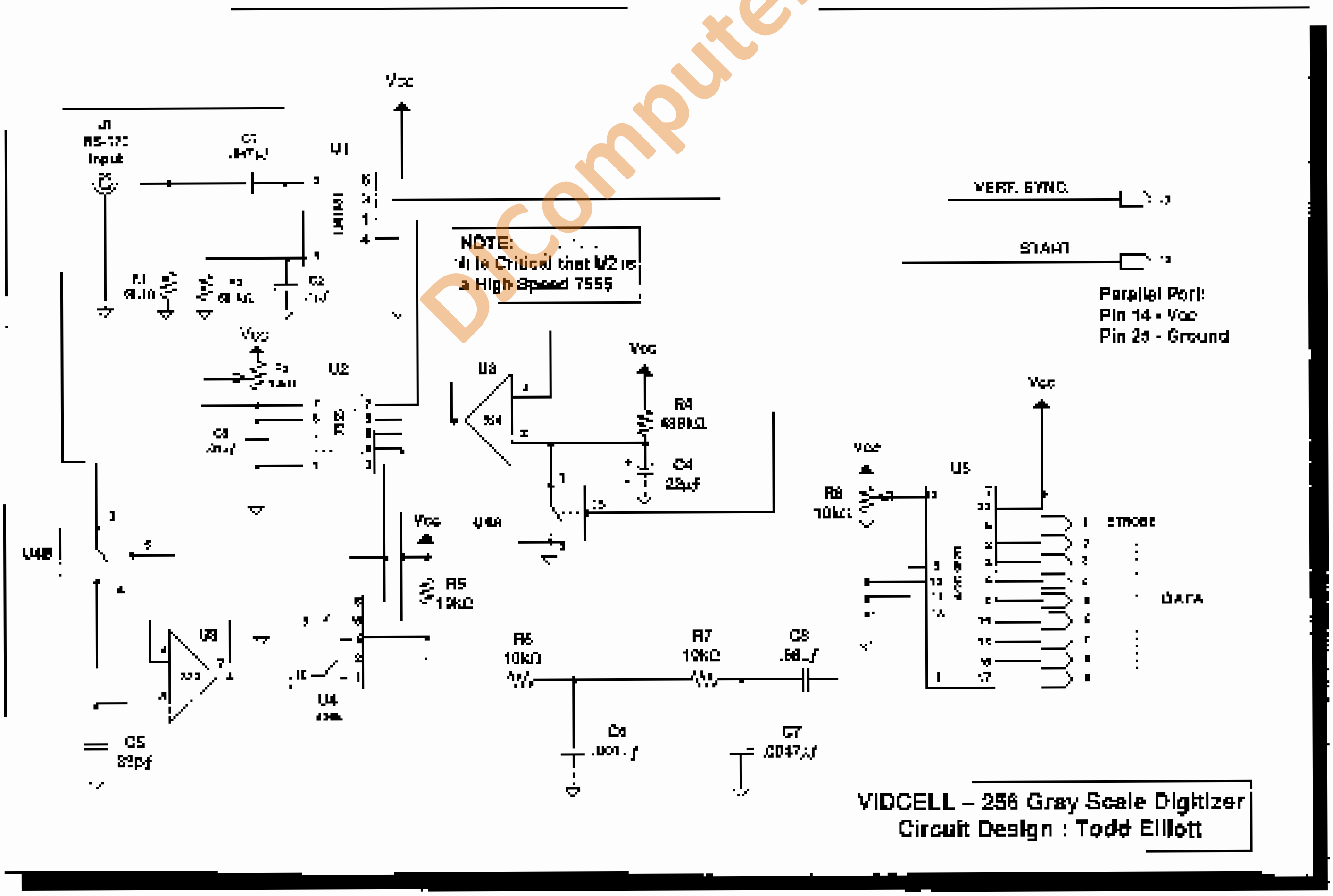
In addition to the parts and circuit board, you will also need a small-tip soldering iron and some quality solder. To aid in the assembly of the components, use a ruler to mark off a distance of .3 inches on a piece of paper. Then, use this as a guide for bending the leads of the components to the right width.

Start by putting the resistors in the board first, followed by the capacitors (smallest first, largest last). Be sure you observe the polarity of C1, C4, C8, and C9. If you ordered the exact parts listed, then two of your IC sockets have built-in filter capacitors. The 14 pin is for U4 and the 20 pin is for U5. As you insert the sockets in the board, you may find that bending the corner pins a little helps to keep the socket in place for soldering. R3 and R8 are both 10k potentiometers. After soldering potentiometers R3 and R8, adjust them both to the center position. R3 controls fine width adjustment and R8 controls image contrast. Once the circuit is operational, these can be adjusted to make the digitizer work better with each specific camera.

C9, C10, and C11 are not shown in the schematic. C10 and C11 are both .1uF filter caps and C9 is a 220uF electrolytic filter cap, as shown in the parts list. Make sure all leads are cut as short as possible after soldering. CN1 is the DB25 connector that connects the unit to the parallel port on the Amiga. You can use a right-angle connector or a straight connector (if you order the parts from me, please specify which type of connector you would prefer for CN1). The next and final step in construction is to connect the RCA jack. This is done by soldering some wire to J1. The positive or inner connection on the jack should be soldered to the hole closest to the bottom of the board, or to the right if you are looking at the J1 label right side up. Clean off any excess flux with a non-organic flux remover (be sure to read the warning label on this stuff). Insert all ICs in their sockets, making sure they are oriented correctly. Your new digitizer should now be ready for action.

MAKING IT WORK

After double checking all parts and connections, turn the Amiga off and plug the digitizer into the parallel port. Turn the power on and boot up the software called VidCellv1.0, if you have at least 1 meg of memory. If you have 512K, then use the VC1.0 software. Next, connect a video source such as a camera or a VCR to the input jack J1. The best results can be obtained with a high-resolution, black-and-white camera such as



VIDCELL – 256 Gray Scale Digitizer
Circuit Design : Todd Elliott

the Panasonic WV1410. Since the digitizer is not real time, the video source must remain stable for about 10 seconds. Click on the GrabFrame gadget to start digitizing if you are using VidCellv1.0, or the M-key if you are using VC1.0. The screen will go blank. After a short pause, the power light should start flashing rapidly. Each flash represents a vertical sync signal. After the screen comes back, the title bar will display the current phase of processing before display. After a few seconds, you should see an image appear on the screen. You may have to adjust the lighting and focus several times before getting it right. Once you are happy with the display, you may make some fine adjustments on the brightness and contrast with the software before saving your image as an IFF file. You will find that very fine adjustments are possible with 256 greyscales in memory. Documentation and the assembly language source code for all the software's features can be found on the disk with the software.

CONSTRUCTION COMPLETE!

A lot of time and effort has gone into making this project a useful, professional quality project for Amiga users and programmers. Hopefully, having the source code available will inspire you to find some new applications and improve it even more.

Write me and tell me what you think of the project and how you think the software could be improved in the future. Also write me if you have any problems with it, and I will try to help the best I can.

PARTS LIST

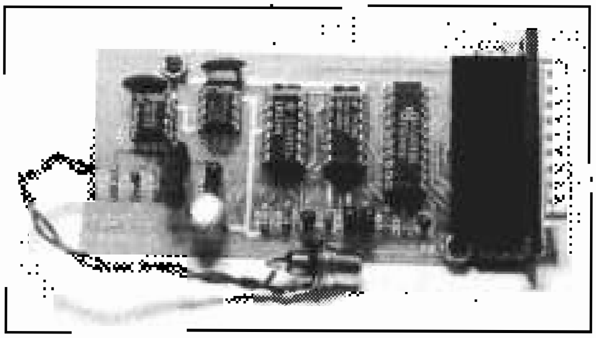
Qty	Part Reference	Digi-key Part	Desc.
1	U1	LM1881N	SYNC SEPARATOR
1	U2	LMC555CN	CMOS 555 TIMER
1	U3	LM324N	QUAD OP AMP
1	U4	CD4066BCN	QUAD SWITCH
1	U5	KAD0820BCN	8 BIT A/D CONVERTER
1	R1	68.1X	68.1 OHM RES
1	R2	681KX	681K OHM RES
2	R3,R8	QOG14	10K OHM TRIM POT
1	R4	499KX	499K OHM RES
3	R5,R6,R7	10.0KX	10K OHM RES
1	C1	P2099	.047 UF POLYPROP CAP
1	C2	P3104	.1 UF POLYPROP CAP
1	C3	P3103	.01 UF POLYPROP CAP
1	C4	P2028	22 UF TANT CAP
1	C5	P4019	30 PF DISC CAP
1	C6	P3102	.001 UF POLYPROP CAP
1	C7	P3472	.0047 UF POLYPROP CAP
1	C8	P2072	.68 UF TANT CAP
1	C9	P6002	220 UF RAD ELEC CAP
2	C10,C11	P4311	.1 UF DISC CAP
1	CN1	325M-ND	DB25 MALE RIGHT-ANGLE
2ND OPTION			
1	CN1	525M-ND	DB25 MALE STRAIGHT
2	S1,S2	C7208	MACHINE 8 PIN SOCKET
1	S3	C7214	MACHINE 14 PIN SOCKET
1	S4	ED2101	14 PIN SOCKET W/CAP
1	S5	ED2104	20 PIN SOCKET W/CAP
MISC			
1	J1	RADIO SHACK # 274-852	RCA JACK
1		RADIO SHACK # 270-257	CASE (OPTIONAL)

USING VIDCELL V1.0 SOFTWARE WITH THE VIDCELL DIGITIZER

FROM THE BEGINNING

The idea for this project first came to me more than a year-and-a-half ago; it has basically taken that long to get it ready to go. I started by just drawing up a schematic, breadboarding and discovering the circuit wouldn't work! This went on almost every night (sometimes until 4 in the morning) for about a month, until it finally did work. It didn't look great at first, either, but within a couple of days I had it looking pretty good, for a breadboarded circuit. The joker.pic on the disk was one of the first digitized pictures I saved.

Once the bugs had been worked out of the circuit, I purchased Pro-Board from Prolific Inc. and designed the circuit board in a weekend. It worked, but I had to make changes. The software still had a ways to go, too. I needed a file requester. I searched through the public domain until I found R.J. Mical's file requester (thanks, R.J.!). Once I felt that the software had reached a point that it could be released, I wrote this article, researched the best components to use, and maintained a social life again. It has been much fun, and I hope a lot of Amiga people really enjoy having a video digitizer with a schematic, theory of operation, and source code (yeah!). I also hope a few of you out there will eventually do things to make it even better. If you do make some modifications or have ideas for same, please send them to me.



GENERAL INFORMATION

Lighting

I've found that digitizing something isn't as easy as you might think. Lighting is a very difficult thing to get right. Natural sunlight provides the best results, but not everyone wants their computer outdoors! Generally, fluorescent light is very good for luminance, but it doesn't bring out the green colors very well. I have my best luck if I shut off all the lights in a room except for a fluorescent light and an incandescent light illuminating the image to be digitized. Watch for glare and learn to use the *PSEUDO COLOR* option on the color menu (see operation instructions below). Once you get the display looking pretty good (good contrast), make some fine adjustments using the software gadgets before saving your image. I think you'll find that you can make very subtle changes when you have 256 grayscales to work with in memory.

LIB Files: (important!!)

There is a file called requester.lib that must be in the LIBS directory of the system disk for the vidcell v1 program to work.

Raw File Format: (for programmers' information)

The RAW file format contains a byte for each pixel of the image. The information is stored in vertical columns from top to bottom, left to right. So, if the image size is 640 x 200, then there are 640 times 200 bytes, or 128,000 bytes of raw information. The first byte contains the brightness information for the upper-left hand corner pixel, the second byte contains the info for the pixel directly below that, etc., and the 201st byte would contain the information for the pixel directly to the right of the first pixel, the 202nd byte would be the pixel below that, etc. I hope this makes sense.

Memory

If you only have 512K, use the software called VCV1 in the VC directory. This software will take over the machine, but it does allow you to digitize in all the modes. The brightness keys do not work yet on this version of the software. The save option saves an IFF file called "test.pic" in the current directory, and exits.

If you have 1 meg or more, then use the software called VidcellM. This is multitasking software that only takes over the machine when it is accepting information from the digitizer. It has many features that VCM doesn't have. I strongly recommend having *at least* 1 meg of memory for digitizing, and you really need *more than* 1 meg for 640 x 400 images.

USING THE VIDCELL SOFTWARE

The software was designed such that your display is placed in the background with a control panel in front of, but not entirely covering, the display. The control panel can be toggled out of the way by double clicking the right button. The menu options are located on the menu strip of the control panel.

PROJECT MENU

LOAD

This allows you to load a previously saved RAW file and adjust it before saving it as an IFF file. The RAW file format contains the 256 grayscale information, while the IFF does not.

SAVE

RAW saves the current raw information since the last GrabFrame was activated.

IFF saves the current display that you are viewing as an IFF image that can be used in other software packages, such as DeluxePaint and PIXmate.

QUIT

This exits the program, obviously.

MODES MENU

COLOR

RED - shows the display in shades of red (for use with color filters).

GREEN - shows the display in shades of green.

BLUE - shows the display in shades of blue.

MONOCHROME - shows the display in shades of gray.

PSEUDO COLOR - shows the display in order of intensity. This helps to determine problems with the lighting, such as glare and shadows in certain areas. The order from dark to light is: blues, reds, greens, yellows, white.

SCREEN SIZE

320 x 200 - sets the display mode to 320 x 200 pixels (aspect ratio not right).

320 x 400 - sets the display mode to 320 x 400 pixels (aspect ratio not right).

640 x 200 - sets the display mode to 640 x 200 pixels (default).

640 x 400 - sets the display mode to 640 x 400 pixels (requires more than 1 meg to be reliable).

PLANES

Unfortunately, this is not available in this version.

GrayScale

This option sets the display mode to shades of gray.

BlackandWhite

This option sets the display mode to B&W for use with desktop publishing. It is used in conjunction with the threshold gadget (T) to set the lightness/darkness.

CONTROL PANEL OPTIONS

I - This gadget is the intensity gadget. It allows you to change the brightness of the picture. To use it, just set the percentage where you want it and select the REMAP gadget. You will see the current phase of processing in the menu strip before it actually displays the changes.

C - This gadget is operated the same as the I-Gadget, but it controls the contrast of the image.

T - This gadget is used in conjunction with the Black and White mode, selected from the menu. It allows you to change the light/dark features. It works very well for creating images to be used with desktop publishing.

GrabFrame - This gadget is what actually starts the digitizing process. It takes over the computer temporarily and blanks the screen. When the screen comes back, you see the current phase of processing before the actual display is updated.

Reset - This gadget resets the I, C, T gadgets. It opens the WorkBench (if it was opened when the program was started). It sets the display mode to 640 x 200 with monochrome.

Negative - This gadget creates an exact negative of the current display. It doesn't affect the RAW information.

ClearScreen - This gadget clears the display. It doesn't affect the RAW information.

Smoothing - This gadget (when illuminated) uses an averaging routine on the data to smooth out image edges. It gives the effect of blurring the image.

Dithering

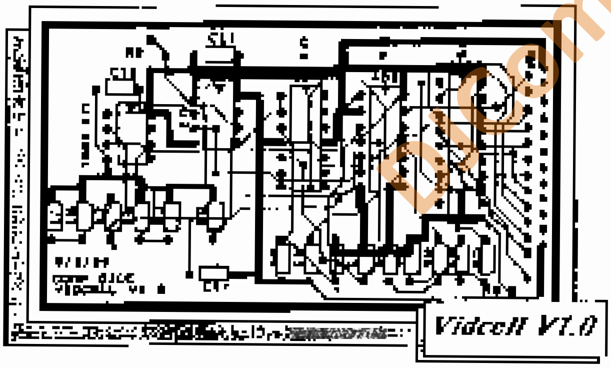
Not available in this version

Screen Position

Not available in this version.

I guess that about covers the software for now. If you have suggestions or questions, please feel free to write.

☑



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 here
or call 1-800-345-3360

Next issue of ***AC'S TECH*** available April 1991.

An Introduction to Interprocess Communication with ARexx

by Dan Sugalski

One of the more useful, and least understood, capabilities of ARexx is its ability to communicate with other programs running simultaneously. Unfortunately, the full capabilities of the language are almost never used. Part of the problem is the ARexx manual is set up as a reference work rather than a tutorial, and part of the problem is a lack of good examples. This article will provide you with both tutorial explanations and some clear examples.

Almost everyone who uses the language is familiar with some of the communications facilities provided in ARexx. The main selling point of the language is its ability to integrate with a variety of programs and provide a standard macro interface. Doing this requires ARexx to talk to programs, and vice versa. Unfortunately, this communication is almost all one way.

Surely you are familiar with the way ARexx is most commonly used. You write a macro for your communications program or text editor that does something useful. Whenever that macro gets "fired up" it gets passed some parameters, does some processing, and then fires off a series of commands to the host that called it. While this is certainly handy, it is also limiting. Once the macro begins running, the only information it ever gets back from the host program is an occasional status code.

This is unfortunate, because not only can ARexx send commands to other programs—it can also receive them. Two-way communication opens up a whole new world of possible programs. In the old way of doing things, ARexx programs were mostly stuck dealing with only one host. Sure, ARexx has always been able to talk to more than one host, but so what? Do you want to integrate your text editor and communications program so you can use it (instead of the clunky message editor on BIX or your favorite BBS), but think it's impossible? And how about using a paint program to edit bitmaps interactively for your DTP package?

Having the ability to send and receive information makes the impossible possible. There are a number of programs that are only marginally useful with the old one-way techniques, and a few programs that are almost completely useless. The question is, how is it done?

All of ARexx's communications functions are built upon the message-passing system that is the heart of the Amiga's operating system. To use ARexx effectively requires at least a certain amount of familiarity with these functions, so hold on for a whirlwind tour of what is inside of your Amiga.

All the information exchanged between programs and the Amiga operating system is done by using messages and message ports. An analogy would be to think of them as letters and mailboxes. With a letter, you write it, put your return address and the address of the receiver on it, and entrust it to the post office. Things work similarly with messages. A program creates a message, puts its return address on it, and gives it to the message system, along with the address of the message port it is supposed to go to. To help simplify things, all message ports have a name that the system keeps track of. So, rather than having to try to figure out where a message port is in memory, your program can ask the system for the location of the message port with the name "Fred".

There are a few differences, of course, between mail and Amiga messages. Unlike mail boxes, a program may add and remove message ports from the system. Also, all messages are taken out of message ports in the order they were received. Most importantly, all messages must be returned to the program that sent it. Continuing our analogy, it is kind of like only borrowing your mail, rather than keeping it.

To help manage messages and message ports, the OS provides a number of useful capabilities. The most important is the ability to put your program to sleep until a message arrives at one of its message ports. The only alternative to doing this is constantly checking your message ports for new messages, something that is wasteful of precious CPU time that can be better used by another program.

Now that we have had a quick overview of the underlying message system ARexx uses, let's turn to the details of how it is used. The simplest of the functions are the two used in nearly all ARexx programs: the ADDRESS command and the 'Host Commands'.

The ADDRESS command is the most straightforward. It tells ARexx the case-sensitive name of a message port. Whenever your program needs to send out a message, it goes to the port named by the last ADDRESS command executed. Note that ARexx doesn't check to see if the port exists until it has to send a message out. In addition, Amiga message port names are case sensitive; that is, ARexx upper-cases all text that isn't enclosed in quotes. This means your program may read 'address Fred', but when ARexx actually executes the line it sees 'ADDRESS FRED', two entirely different things.

A "Host Command" is pretty much anything ARexx doesn't understand. When the ARexx interpreter is running your program and comes across a line that isn't in the ARexx language, it takes the line, substitutes the value of any variable for the variable itself, packages it up in a special ARexx message, and fires it off to the port your program last ADDRESSed. ARexx then puts your program to sleep, until that message gets a reply.

That last detail is important, and something that must always be kept in mind. If the message your program sends out never gets a reply, your ARexx program will never wake up. This also means your program can't send messages to itself. A message will sit at your program's port until it is received and replied to, but your program can't get the message because it is asleep waiting for it to reply to itself. I'm sure you can see the problem involved here.

Besides being able to send messages, ARexx programs also have the capability to receive them. There are a number of functions available to manage messages and message ports in the RexxSupport library that comes with the ARexx interpreter. There is a brief reference section covering this library in Appendix D of the manual that comes with the language, but we will be going over the functions of interest in more detail. One thing to keep in mind when reading the manual is that ARexx refers to messages as "packets", and the data in the messages as "arguments". This can occasionally lead to confusion, so be careful. Before doing anything with messages, ARexx must have access to the support library. To do this, insert the line "CALL ADDLIB('RexxSupport.library', 0, -30, 0)". This makes sure the library is loaded into memory and is ready for ARexx's use. As with all files, make sure the library name is spelled correctly and has the ".library" extension. Once you have done that, your program is ready to cope with messages.

To fully use messages, your ARexx program has to be able to do a number of things. Among them, it must

(1) open and close message ports
(2) send messages
(3) receive messages
(4) reply to messages
(5) wait for messages
(6) get data out of messages

We've already seen how to send messages, and the support library has routines to do everything else. Before receiving any messages, your program must open up a message port. The OpenPort(port name) function causes ARexx to open up a message port with the specified name for your program. This name must be unique, and should be upper case. ARexx will return a 1 if the port is opened successfully, and a 0 if it is not. Note that the manual incorrectly states that you are returned the address of the message port.

Once your port has been created, your program has to wait for messages to arrive. To do this requires the WaitPkt(port name) function. This puts your program to sleep until a message is at the message port. When the port is no longer empty, your program wakes up and continues executing at the statement after the WaitPkt( ). If there is already a message at the message port, your program will never go to sleep.

(There is a bug in the support library for version 1.06 and before. In these versions of the library the WaitPkt( ) will only wake up if a new message arrives. If there is already a message at the message port WaitPkt( ) does not recognize it, and waits until a new message arrives before it wakes up.)

With WaitPkt( ) awake, your program can get any messages from its message port. To do this, call GetPkt(port name). This takes the first message out of the message port and returns the address of it to you. If there are no messages at the port, the return value will be NULL ('0000 0000'x). Your program should always check to make sure it gets a non-NULL address for all messages. Your program may occasionally get woken up even if there are no messages available yet, and the system returns a NULL value if you try to get a message from an empty message port. Doing anything with a message pointer equal to NULL is a good way to get to know the Guru better.

After your program has successfully gotten a message, it has to extract the information from it. Each ARexx message can contain up to sixteen strings of characters, though typically there is only one. To get at this data, your program must use the GetArg(message address, argument slot) function. This will extract one of the strings, numbered zero through fifteen, and return it to your program. If the slot number is omitted, then the value in slot zero is returned. This is one of the few calls that requires your program to recognize the kind of data it is getting in the message, because a call to GetArg( ) with an empty slot number will abort your program with an error message.

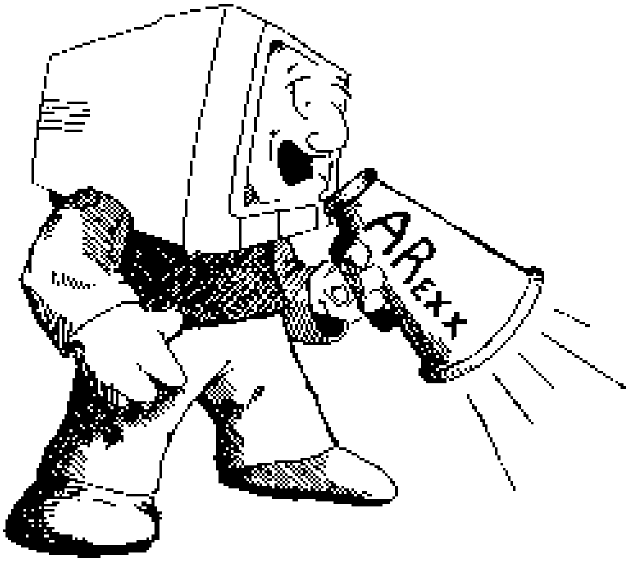
Now you've gotten a message and retrieved whatever data you want out of it. Whenever you've finished with a message, your program should reply to it and give it back to the program that sent it. The Reply(message address, return code) function will do this for you. If you want, you can return an optional return code. This must be a positive integer, though its meaning is entirely up to the program that sent the message.

Once you have completed all this processing, it's time to go back to the beginning and do it all over again. When your program is finished with its port, it should call the ClosePort(port name) function. This closes your message port and automatically replies to any messages still in it. While supplying this function is not absolutely necessary, as ARexx will close any port you have open when your program exits, to do so is a good habit to get into, and certainly can't hurt.

Well, that concludes our quick run down of the functions you need to pass messages back and forth. Perhaps at this point you are wondering, "How useful can this be?" After all, most programs that have ARexx capability don't have any provisions for sending messages to ARexx macros already running.

This is where things get interesting. ARexx's message capabilities have one limitation ... they can only cope with a special message format, called, oddly enough, a RexxMsg. These are the only messages you can do a GetArg( ) on. Conveniently, though, these are the types of messages that ARexx sends out. The net effect is that one ARexx macro can exchange messages with another.

The two sets of example programs use this fact to show, I hope clearly, exactly how it works. The first two programs are a matched set, "Sample1.rexx" and "Sample2.rexx", and they should be in your REXX: directory. They each do very little, to



keep the concepts as clear as possible. Simple1 starts up Simple2 and sends it messages. Simple2 receives the messages and prints out the contents of the passed string.

Simple1 starts out with the mandatory comment, starts up trace mode, and tells ARexx to talk to AmigaDOS with the ADDRESS statement. Next, it opens up the support library and starts up Simple2 to receive its messages. It then waits a bit to give Simple2 a chance to start up.

At this point, Simple2 has started. The first thing it does is open up its message port. Then it goes into its message loop. It waits for a message, gets it, gets the passed argument, and replies to the message. The loop is very simple, of course. There is no error checking or processing of the data, but it should show pretty clearly how things are supposed to look.

Meanwhile, back in Simple1, we address the port opened by Simple2. Once again, there is no checking to see if the port has actually opened. It is possible (in fact, it's a very good idea) to add this capability; it will be discussed a bit later in this article. Anyway, in the main loop of Simple1, you'll notice the line 'i i i'. This is obviously not an ARexx statement, so it is packed up in a message and sent to Simple2's port. When it gets printed out, however, you'll notice that, rather than spewing out more than twenty lines of 'i i i', it first prints '1 1 1' then '3 3 3' and continues on until it stops.

Why does this happen, you ask? Simple. One of the nice features of ARexx is the way it treats variables. Simply put, anything that isn't a legal command, function, or quoted constant is treated as a variable. When ARexx encounters a variable in the course of program execution, it automatically replaces the variable name with the variable's contents. In this case, as we use I as the loop variable, ARexx inserts the value of I in before the message gets sent.

Now, while this is very handy, it does introduce the possibility of subtle bugs. Namely, it raises the possibility that what you think is a string constant is actually a variable with different contents than its name. This following program fragment illustrates this:

```
/* Insert your favorite comment here */
Top = "Hi Mom!"
/* Time passes */
Jump Top
```

You would probably expect that 'jump top' would be sent out in a message. What really gets sent out is 'JUMP Hi Mom!', something else entirely. The best way to avoid things like this is to enclose anything you want sent out literally in quotes. ARexx won't touch anything within quotes, so it's best to use them liberally whenever possible to help cut down on the possibility of bugs. ARexx also automatically upper-cases everything that isn't quoted, something else you may not intended to have happen.

In anything other than the trivial example programs we just went over, there is going to have to be at least some error checking. Besides doing things we already discussed, like checking the status code returned by OpenPort() and the validity of the address of messages, it is always a good idea to make sure the port we want to talk to exists. When dealing with other programs, it is never safe to assume anything. Checking to see if a port exists is a simple operation. In fact, ARexx has two functions that can check for message ports - Show() and ShowList(). The latter, ShowList(P, port name), resides in the RexxSupport library and returns a 1 if the named port exists and a 0 if it does not. Show(P, port name) is in the base library of functions and performs identically to ShowList(). The major difference between the two is that Show() checks the lists that ARexx keeps and only returns port names opened in ARexx programs, while ShowList() examines system lists and will check to see if any program in the system has opened up the named port. Show() only works if you have ARexx version 1.10

Listing One
Simple1.rexx

```
/*simple1.rexx*/
trace r
address command
call addlib("rexxsupport.library",0,-30,0)
run rx simple2
call delay(100)
address PORT
do i=1 to 50 by 2
    i i i
    end
exit
```

Listing Two
Simple2.rexx

```
/*simple2.rexx*/
call openport(PORT)
call delay(300)
do i=1 to 10 by 2
    call waitpkt(PORT)
    pkt=getpkt(PORT)
    say getarg(pkt,0)
    call reply(pkt,0)
    end
call closeport(PORT)
exit
```

or above, so it is best to use ShowList() if possible.

Now that we have all the pieces to build a properly working message handling system into an ARexx program, lets look at a more complex example than the last. Programs three and four show a working set of programs. These use Willy Langeveld's RexxARPLib library to access Intuition's gadget and drawing routines. Make sure you have version 2.3 or greater of the library, as the circle drawing function isn't available in earlier versions. The first program takes care of all the user interaction, while the second does all of the drawing. Specialization like this, while a bit inefficient, makes programming easier. Each program has to deal with a minimum number of things, allowing it to be smaller and easier to understand, and much easier to debug.

Graphics1, the main program, first uses the ADDRESS command to prepare to issue commands to AmigaDOS. Next, it checks to see if the libraries it needs are already loaded; if not, it loads them. Then, it spawns off a small program that uses the RexxARPLib to do all the actual communicating with Intuition. ARexx programs use this as an interface to perform all the graphics and windowing routines we need. Each window a program needs to have requires a separate CreateHost call.

CreateHost is the most peculiar function in the RexxARPLib library. What it does, once called, is to take over the process that called it. Its first parameter tells it the name of the message port it should create and listen to, and the second tells it what message port to send its messages to. Most of the remaining functions in the library perform their various graphics functions by building messages and sending them to a CreateHost created message port.

After we make the CreateHost call, the program goes into a short loop that checks to see if the port has been opened. If not, it waits a second and tries again. If, after ten seconds, the port still hasn't been created, the program prints out an error message and exits. Next, we start up the task that does all the drawing, then open the port that will be getting the Intuition messages.

The next few calls set up the window and gadgets that are used to interact with the program. We create two gadgets, labeled 'Box' and 'Circle', and have Intuition attach CloseWindow and drag gadgets to the title bar. Our program will receive a message with "CLOSEWINDOW" in slot zero every time the CloseWindow gadget is hit. The actual closing of the window is completely under the program's control.

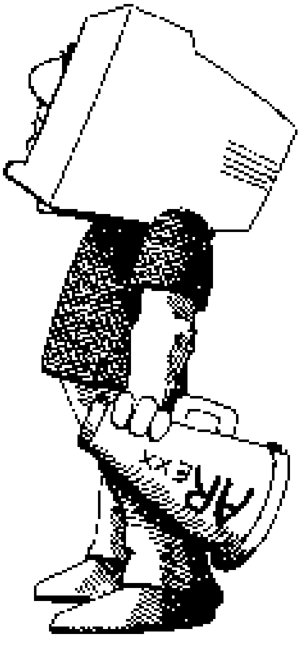
At the same time the first program is opening up its window and attaching its gadgets, the second program is starting up. At this point, the first program goes into a loop, checking to see whether the second program has opened up its message port. If—again after ten seconds—it hasn't, we close our window and message port, send a message to the user, and exit with an error code.

If the second program has started successfully, the first program sets itself up to talk to it. What follows is a fairly standard message loop, the type which you'll become very familiar with after one or two ARexx programs. It waits for a message from the window we opened. When it gets one, it extracts the data out of it and replies to the message. The contents of the message is echoed out to the CLI window the program was started from, so you can see exactly what it got. That data is then sent to the drawing program so that it can perform whatever processing needs to be done. The final step in the loop is to check to see if the message we got is telling us to close up shop. If so, we do; if not, the loop starts all over again.

The second program is a lot like the first. It also checks to see if the libraries it needs are open, and if they aren't, then it opens them. It, too, starts up a CreateHost process to manage the drawing window, and checks to see if it is created successfully. As we pass a message port name to CreateHost, we are guaranteed that, with no gadgets in our window, we will never get any messages from it. The program then opens its message port and window.

The message handling loop in this program is quite similar to the loop in the first program. In this program, however, there is a bit more in the way of processing that needs to be done for each message that arrives. When we get a valid message we first extract the data from it, then echo that data to the CLI, and reply to the message. What follows next is a series of checks on the passed data. We can be told to draw a circle, draw a filled box, or close up. The circle routine chooses a random radius, X and Y coordinates, and color. We set the drawing pen to the color we just chose and draw our circle outline. Unfortunately, there is no simple routine available to draw filled circles, only outlines. The box routine is similar. Random X and Y coordinates are chosen for the upper left and lower right corners of the box, along with a random color. The program sets the color and draws a filled rectangle. The CloseWindow routine closes up the window and message port and exits the loop. After the loop, the program exits with an error code of zero, indicating a normal exit.

As you can see from the two examples, adding message support to your ARexx programs is almost trivially easy, requiring only a few simple ARexx commands and function calls. While proper use of the message facilities does take a little thought, the actual routines themselves are straightforward. You should find that, after a little while, you are writing programs that can glue together many different applications in customizing your Amiga environment to work better for you.



Listing Three
Graphics1.rexx

```
/* Graphics1.rexx */
/* trace r */
address command
if ~show('L',"rexxsupport.library")
    then call addlib("rexxsupport.library",0,-30,0)
if ~show('L',"rexxarplib.library")
    then call addlib("rexxarplib.library",0,-30,0)
run rx "call createhost(THEPORT,THEPORT2)'"
do i=1 for 10
    if showlist(P,'THEPORT') 0
    then call delay(50)
    else break
end
if showlist(P,'THEPORT') 0
    then
    do
        say "It didn't work!"
        exit 10
    end
run rx Graphics2.rexx
dummy=openport('GRAPH1')
call openwindow('THEPORT',10,10,100,90,'CLOSEWINDOW+GADGETUP','WINDOWCLOSE+WINDOWDRAG','Test')
call addgadget(THEPORT,30,15,1,'box',BOX)
call addgadget(THEPORT,20,30,2,'circle',CIRCLE)
do for 10
    if showlist(P,GRAPH1) 0
        then call delay(50)
end
if showlist(P,GRAPH1)=0
    then do
        say "Can't start second task!"
        call closewindow(THEPORT)
        call closeport(THEPORT2)
        exit 10
        end
address GRAPH1
do i=1 forever
    call waitpkt('GRAPH1')
        p1 =getpkt('GRAPH1')
        if p1 ='0000 0000'x then
           iterate
        mparg = getarg(p1)
        call reply(p1)
        say ' ' mparg ' '
        mparg
        if mparg == CLOSEWINDOW
            then leave 1
        end
    end
call closewindow('THEPORT')
call closeport('THEPORT2')
exit
```

Listing Four
Graphics2.rexx

```
/*Rexx prog #1. Gets messages and draws figures from them */
address command
/* trace r */
if ~show(L,'rexxsupport.library')
    then call addlib('rexxsupport.library',0,-30,0)
if ~show(L,'rexxarplib.library')
    then call addlib('rexxarplib.library',0,-30,0)
"run rx "'call createhost(GRAPH,GRAPH1)'"
do i=1 for 10
    if showlist(P,'GRAPH1')=0
        then call delay(50)
        else leave
end
if showlist(P,'GRAPH')=0
    then exit 10
call openport(GRAPH1)
call openwindow(GRAPH,130,105,450,105,,"WINDOWDRAG","Output Window")
do q 1 forever
    call waitpkt('GRAPH1')
    do i 1 forever
        pkt getpkt('GRAPH1')
        say 'Got Packet!'
        if pkt '0000 0000'x                /*No message, so we wait again */
            then leave i
        mycommand getarg(pkt)
        say '  ' mycommand '--'
        call reply(pkt)
        if mycommand = CIRCLE
        then do
            radius=random(1,49,time('S'))
            x=random(1+radius,399-radius)
            y=random(1+radius,99-radius)
            color=random(1,4)
            call setapen(GRAPH,color-1)
            call drawcircle(GRAPH,x,y,radius)
            end
        if mycommand   BOX
        then do
            x=random(1,398)
            y=random(1,98)
            x2=random(1+x,399)
            y2=random(1+y,99)
            color=random(1,4)
            call setapen(GRAPH,color-1)
            call rectfill(GRAPH,x,y,x2,y2)
            end
        if mycommand == CLOSEWINDOW
        then do
            call closewindow(GRAPH)
            call closeport(GRAPH1)
            leave q                        /*exit outermost do loop */
            end
        end
    end
exit 0
```

An Introduction to the ilbm.library

by Jim Fiore, dissidents
BIX: jfiore

Back in 1985, Electronic Arts introduced the Standard for Interchange Format Files, or as it has come to be known, IFF. Several different file types (called FORMs) were described, including those suitable for text (FTXT), midi-fi audio samples (8SVX), and music scores (SMUS). The one variant which has made the biggest impact on the Amiga community is the InterLeaved BitMap, or ILBM, FORM. ILBM has become the undisputed standard for Amiga bitmap-type graphics files, (literal computer screen imagery versus the vector-type structured drawings found in CAD or professional illustration programs).

It would be unthinkable for a modern paint or image manipulation program not to support the import and export of ILBM files on the Amiga. Thanks to this level of standardization, Amiga users need not keep track of myriad file conversion utilities which so often plagues the users of other systems.

As a developer or programmer, it is obvious that the survival and usefulness of your applications are tied to the IFF standard. What is not so obvious is the amount of work which is required in order to read and write these files properly. Initially, if you needed to create an ILBM reader/writer, you had two choices: hack apart the Electronic Arts C code for your application (and language of choice, if not using C), or write your own routines from scratch. The problem with the second choice is that if you don't use a full implementation you run the risk of only being able to read files written in a specific manner.

The problem with the first choice was immediately apparent to non C literate programmers. Even for established C users, there was quite a bit of code to wade through plus a few bugs as well. This boiled down to quite a bit of work for the simple concept of allowing the user to save a given window as an IFF ILBM file, so that the image might be imported into a desktop publishing or paint package. Besides, it's wasteful to place this code into every application which needs it.

A standard Amiga shared library, which any application can open and use, is a much more efficient approach. In the ideal world, there would be standard system libraries which would handle both high level and low level calls for reading and writing IFF files. The 2.0 version of the operating system does offer iffparse.library, which is designed to handle the low level calls, although it does not offer high level, FORM specific calls. Given our own needs here at dissidents, the ilbm.library was born.

The ilbm.library was created by Jeff Glatt, and offers low level and mid level general IFF calls, along with high level ILBM-specific calls. The concept of the ilbm.library revolves around the original Electronic Arts code. If you already have applications which were written using this code, switching to the ilbm.library will involve a minimum of work since the library contains virtually all of the same functions but comes with a few twists.

Unlike the original EA code, the library is written entirely in optimized 68000 assembly, for small size (less than 7000 bytes) and fast execution. By eliminating the bulk of your application's IFF code, your application will be both smaller and faster. High level calls have been added which make saving and loading ILBMs almost trivial. Although the library is skewed for use with ILBMs, it can be used with any IFF file while special support for ANIMs has also been added. Finally, you have the ability to insert custom handlers for various aspects of the IFF file.

In spite of its power and ease of use, ilbm.library does not pretend to be everything for everyone. Programmers with very special needs may still prefer to write all of their own routines from scratch. For the vast number of people who need a relatively painless way to deal with ILBMs in more general ways, ilbm.library can save considerable development time.

If you find the library useful, you may use it in any of your applications, be they commercial, shareware, or otherwise. ilbm.library and its associated documentation and examples are properly referred to as FreeWare. They are not public domain, since the author still retains the copyright. Consequently, you cannot sell the library as a distinct product, or represent it as

For the vast number of people who need a relatively painless way to deal with ILBMs in more general ways, ilbm.library can save considerable development time.

such. Short of that, there are no licenses, fees, royalties, or other forms of rabid capitalist trickery to deal with. As a matter of fact, you don't even have to tell your users where ilbm.library came from. Also, since this is a standard shared (versus link) library, you get to use it from the language of your choice.

It would be impossible to write a single article covering all of the functions and aspects of ilbm.library. Indeed, the Doc file alone is some 67K bytes in length. You will find this Doc file, along with examples in C and assembly, (with BASIC notes) on disk. These examples include a picture viewer, an IFF scanner, an ANIM example, and even how to use the library with non-ILBM files (in this case, an 8SVX sample player). In this article, we'll take a look at perhaps the most general application of all: the ability to load and save ILBMs in a C language program.

Our example is called AC_ILBM.c and it demonstrates how you would use the ilbm.library in typical applications. It will allow you to read in and display an ILBM file, in either your own window and screen, or in one which the library will open for you. It will also allow you to alter the colors of the picture (using color.library) and save the result as a new ILBM file. Usage is as follows: the program must be called via the CLI. The second CLI argument directly after the program name will be the name of the file to view. An optional third argument will force the picture to load into a 640 by 200 window which we will open. If the file is some other size it will be scaled to fit. Without the third argument, the proper window and screen for the picture will be determined by the library and opened for us. The library will scan the file, and if there is an error, allow us to print an error message to the CLI. If all goes well, the picture is displayed. The ESCape key will be used to terminate the program, F1 will be used to call up the ColorTool palette so that the picture may be altered, and F2 will save the picture as a new file.

To use ilbm.library, we need to include the ILBM_Lib.h header, and declare a few data items. This includes ILBMBase, the library base pointer. As with any shared library, ilbm.library must be opened before use, and closed upon program exit. We will also need an ILBMFrame. This is a structure which the library functions use to keep track of the file, and also allows you to set certain options. The ILBMFrame must be properly initialized before use (more on this in a moment).

This application uses the library's two high level functions: LoadIFFToWindow( ), and SaveWindowToIFF( ). A third level function, GetIFFPMsg( ), will allow us to extract pointers to error strings which we can then display to the user. In this example, the strings are just printed to the CLI, but they could just as easily appear in an application's title bar or status line, or in a Requester. Initially, the program opens the required libraries and scans the command line arguments to determine the proper mode of operation. If a third argument is present, a default screen and window are opened.

Once this is complete, the iScreen, iWindow, and iUserFlags fields of the ILBMFrame are initialized, and LoadIFFToWindow( ) is called. Notice how simple the call to LoadIFFToWindow( ) is. It only takes two arguments. The first argument is a pointer to a string which holds the name of the ILBM file you wish to load, and the second argument is the address of your ILBMFrame. If the iScreen and iWindow fields are non-zero, LoadIFFToWindow( ) assumes that you have opened a screen and window, and will attempt to load the ILBM file into the window, scaling the picture if necessary. If these fields are initialized to 0, then LoadIFFToWindow( ) will examine the picture file, and open a proper screen and window for you. The addresses of the window and screen will then be placed in the iScreen and iWindow fields, so that you can get to them. This function will return 0 if all goes OK. If there is an error, the return value can be used as the argument to GetIFFPMsg( ) in order to obtain an error string.

The iUserFlags field allows you to customize the treatment of the loaded file. These are the possible flags:

MOUSEFLAG	Make the Intuition pointer invisible.
SCREENFLAG	Hide the screen's title bar.
COLORFLAG	Don't use the loaded colorMap. Preserve the present map instead.
NOSCALE	Don't scale a lower res pic to fill a higher res display.
ADJUSTVIEW	Do overscan if larger than Intuition view.
FORCEPARSE	Continue parsing after ANIM.
ANIMFLAG	Set if its an ANIM file.

The scaling routines in the ilbm.library are suitable for general purpose applications, and have specific limits. For example, they will not remap colors if you try to stuff a four bitplane image into a two bitplane screen, or vice versa. Also, due to the inherent pixel color / position interdependence of the HAM viewmode, scaling may result in some rather odd looking colors for HAM files.

If you need to perform color remapping or HAM scaling, you should use the mid level function LoadILBM( ), which gives you the option of adding custom handlers for the various components (FORMs, PROPs, etc.) Also, it is possible to call the library's ScaleImage( ) function for your own special needs. In contrast to the above, if a third command line argument is not given, we allow the library to open the screen and window for us (the else clause). Note how the iScreen and iWindow fields are initialized to 0. When LoadIFFToWindow( ) returns, we copy the addresses of these items into global pointers for future reference. It is very important to note two things at this point: 1) It is up to the application to close down the window and screen upon exiting, since the ilbm library cannot keep track of these items. 2) The IDCMP of the newly opened window is zero. You should immediately call the Intuition function ModifyIDCMP( ) in order to hear the sorts of messages you're interested in. In the example, we are only interested in RAWKEY events, so thats what we set.

No matter whether we open the window and screen, or let the library do it, we fall into the application function hang_around( ). This just waits the IDCMP and looks for RAWKEY messages. It will do one of three things. If the ESCape key is hit, the program ends and the window, screen and libraries are closed. If F2 is hit, the high level ilbm library function SaveWindowToIFF( ) is called. This function takes two arguments: a pointer to a string indicating the name for the file to be saved, and a pointer to the window to be used. In this case, we just use the default name of RAM:Newpic. If the save is successful, the function returns 0. A non-zero return value indicates that something went wrong, such as not enough disk space. There is a corresponding mid level function which offers a bit more flexibility, called SaveILBM( ). If F1 is hit, a color palette is displayed, allowing you to change the picture's color values. This color palette is created using the dissidents color.library. The call DoColor( ), opens and monitors the palette. It is a full-featured color palette, with your choice of RGB or HSV sliders, a default colorMap, and Copy, Spread, and Undo capabilities. (color library is also dissidents FreeWare.) ColorTool is designed for use with non-HAM viewmodes. ColorTool will open on HAM screens, but the results may be hard to predict and it may appear to do nothing at all.

The example program was compiled and linked using Manx 5.0, but there should be few modifications in order to get it to work under the SAS/Lattice system. For starters, remove the reference to #include "functions.h", and replace it with #include "protos/all.h". Make sure that you link with the ILBM and Color library glue routines. Before running the example, copy both ilbm.library and color.library to your LIBS: directory.

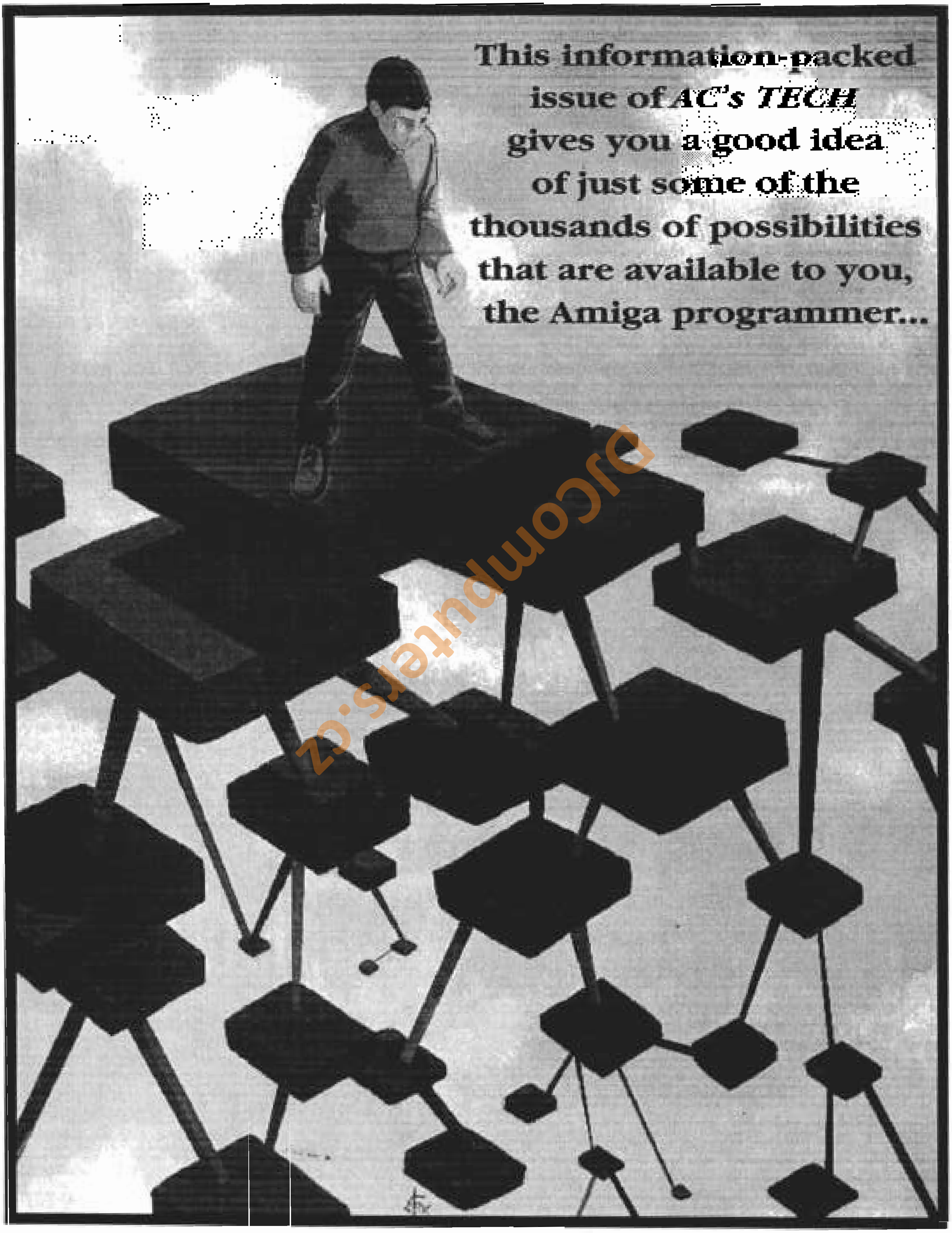
There are many things you can do with ilbm library, and we have only scratched the surface here. There are over 30 basic functions in the library. For more information, I direct you to the documentation and examples on disk. Have fun.

Listing One
ilbm.library
Sample Application

```
/*******************************************************************
   ilbm.library C application for Amazing Computing Tech. Application written
   by Jeff Glatt and Jim Fiore, dissidents, November 7, 1990. Set TAB width
   to 3.

   Using Manx 5.0:

   cc -mcd AC_ilbm.c
   as -cd FileInterface.asm
   as -cd ColorInterface.asm
   ln AC_ilbm.o FileInterface.o ColorInterface.o -lcl

   A note about HAM images: First, ColorTool is designed to work with ordinary
   (non-HAM) screens. You will have some control over HAM screens, but not
   much. Second, while ilbm.library works perfectly well with HAM images under
   ordinary applications, scaling HAM images while loading can produce some
   rather bizarre results. If really need to do something like that, you should
   use the lower level functions to read in the HAM image, and perform the
   scaling yourself.

*******************************************************************/

#include "math.h"
#include "functions.h"    /* Manx's declared lib. functions like proto/all.h */
#include "intuition/intuition.h"
```

```
#include "exec/tasks.h"
#include "exec/types.h"
#include "exec/memory.h"
#include "graphics/gfxbase.h"
#include "graphics/rastport.h"
#include "graphics/gfx.h"
#include "graphics/view.h"
#include "graphics/text.h"
#include "intuition/intuitionbase.h"

#include "ILBM_lib.h"


#define INTUITION_REV 33L
#define GRAPHICS_REV  33L
#define DEPTH          4

struct IntuitionBase *IntuitionBase = 0L;
struct GfxBase       *GfxBase = 0L;
struct Window        *wind_global = 0L;
struct Screen        *screen_global = 0L;

/* Data for the dissidents ilbm.library */
struct ILBMBase      *ILBMBase = 0L;
ILBMFrame             myILBMFrame;

/* Data for the dissidents color.library */
struct ColorBase      *ColorBase = 0L;
extern LONG DoColor();


struct TextAttr my_font_attr = {(UBYTE *)"topaz.font",TOPAZ_EIGHTY,\
                           FS_NORMAL, FPF_ROMFONT};

struct NewScreen ns = {0,0,640,200, DEPTH, 0,1, HIRES, SCREENBEHIND | \
      CUSTOMSCREEN, &my_font_attr, (UBYTE *)"Screen", NULL, NULL };

struct NewWindow nw = {0,0,640,200, -1,-1, RAWKEY, SMART_REFRESH | ACTIVATE |\
      BACKDROP | WINDOWDEPTH | BORDERLESS, NULL, NULL,
      (UBYTE *)"Background Window", NULL,NULL, 640,200,640,200, CUSTOMSCREEN};



VOID open_all(), exit_prog(), bang_around();

VOID main( argc, argv )
LONG argc;
UBYTE *argv[];
{
   IFFP  Result;

   if( argc == 1 )    /* No filename given */
   {
      puts("USAGE: AC_ILBM filename [x]\n");
      exit(0);
   }

   open_all();

   if( argc == 3 )    /* open our own screen, and force pic to scale */
   {
      if( (screen_global = (struct Screen *)OpenScreen( &ns )) == NULL )
         exit_prog();
      nw.Screen = screen_global;

      if( (wind_global = (struct Window *)OpenWindow( &nw )) == NULL )
         exit_prog();
      ScreenToFront( screen_global );

      myILBMFrame.iScreen = screen_global;
      myILBMFrame.iWindow = wind_global;
      myILBMFrame.iUserFlags = SCREENFLAG;   /* hidden title bar */

      Result = LoadILBMtoWindow( argv[1], &myILBMFrame );
      if( !Result )
         hang_around();
   }
   else   /* let ilbm.library create screen and window */
   {
      myILBMFrame.iScreen = 0;
      myILBMFrame.iWindow = 0;
      myILBMFrame.iUserFlags = SCREENFLAG;

      Result = LoadILBMtoWindow( argv[1], &myILBMFrame );
      if( !Result )
      {
         screen_global = myILBMFrame.iScreen;/* save these, since we'll have to */
         wind_global = myILBMFrame.iWindow;     /* close them on our own, later */
         ModifyIDCMP( wind_global, RAWKEY );
         hang_around();
      }
   }

   /* Abort exit through here */
   puts( GetILBMMsg( Result ) );
   damp_mop();
}


/*********************************************************************
                  Opens Intuition, Graphics, Color, and ILBM libs
*********************************************************************/

VOID open_all()
{
   if( !(IntuitionBase = (struct IntuitionBase *)OpenLibrary("intuition.library", INTUITION_REV)) )
      damp_mop();

   if( !(GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", GRAPHICS_REV)) )
      damp_mop();

   if( !(ILBMBase = (struct ILBMBase *)OpenLibrary("ilbm.library", 0L)) )
   {
      puts("Need the dissidents ilbm.library in LIBS:");
      damp_mop();
   }

   if( !(ColorBase = (struct ColorBase *)OpenLibrary("color.library", 0L)) )
   {
      puts("Need the dissidents color.library in LIBS:");
      damp_mop();
   }
}

/*********************************************************************
                  Closes window, screen, libs
*********************************************************************/

VOID damp_mop()
{
   if( wind_global )       CloseWindow( wind_global );
   if( screen_global )    CloseScreen( screen_global );
   if( ColorBase )       CloseLibrary( ColorBase );
   if( ILBMBase )          CloseLibrary( ILBMBase );
```

```
    if( GfxBase )           CloseLibrary( GfxBase );
    if( IntuitionBase )     CloseLibrary( IntuitionBase );

    exit( FALSE );
}


/*******************************************************************
    IDCMP handler. The following RAWKEY functions are implemented:
    ESC:    Quit program.
    F1:     Call up the dissidents ColorTool.
    F2:     Save the picture as "RAM:New.pic"
*******************************************************************/

void hand_events()
{
    struct IntuiMessage *mes;
    ULONG  err_1;

    for(;;)
    {
        Wait( 1 << wind_global->UserPort->mp_SigBit );

        while( mes = (struct IntuiMessage *)GetMsg( wind_global->UserPort ) )
        {
            ULONG       class = mes->Class;
            USHORT      code = mes->Code;

            ReplyMsg( (struct Message *)mes );
            switch( class )
            {
                case RAWKEY:
                    switch( code )
                    {
                        case 0x45:  /* ESCape = quit */
                            clean_up(1);
                            break;

                        case 0x50:  /* F1 = ColorTool */
                            err_1 = DoColor( 0, screen_global );
                            if( err_1 < 0 )
                            {
                                DisplayBeep( screen_global );
                                puts( "ColorTool load error" );
                            }
                            break;

                        case 0x51:  /* F2 = Save */
                            if( SaveWindowToIFF( "Ram:New.pic", wind_global ) )
                                puts("Save error");
                            else
                                puts("Save OK");
                            DisplayBeep( screen_global );
                            break;
                    }
                    break;
            }
        }
    }
}

/*************************** [illegible] END *******************************/
```


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...

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)

Introducing the Premiere AC's TECH Disk

A few notes before you dive into the disk!

- You 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 many 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 ?*

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 869
Fall River, MA 02720-0869

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 utilize low-level disk access. The entire liability for the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc., their distributors, or their 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 archival/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. PiM Publications or any of its agents is not responsible for any damages incurred while attempting this project.

Developing a Relational Database in C, Using dBC III

— by Robert Broughton—

dBC III, a Lattice product, is a set of C callable functions which read and write disk files in the dBASE "standard" format. It is a mature product, having been available for at least three years and it is valuable to Amiga software developers because it provides the capability for reading and writing disk files with random access, using keys rather than byte displacements. The best way to illustrate this is with an example. Here is a C structure which defines a record in a file containing information about sporting goods:

```
struct SportsRecordLayout
   {
   sport      char[15];
   item       char[17];
   material   char[10];
   weight     char[5];
   vendor     char[30];
   price      char[7];
    } SportsRecord;
```

Note that although "weight" and "price" are obviously numeric values, they are defined here as "char"; this will be explained later.

dBC III provides a function, dBgetr, which can read a record from a file like this by supplying the record number. However, any file processed by dBC III can have one or several "indexes" associated with it. In this example, you may want to see all items used for a particular sport; if you supply the key "baseball", you would expect to retrieve records with "bat", "ball", "infielder's glove", "catcher's mitt", etc. in the "item" field. To do this, the "sport" field would be set up as an index. You might also want to retrieve a specific piece of equipment, such as a baseball bat. This can be done by defining "sport+item" as an index. You may want to see all the sports in which bats are used (baseball, softball, and cricket), so "item" can be defined as an index. You may want to see all of the items provided by a particular vendor (Louisville Slugger also makes hockey sticks), so "vendor" could be an index.

A couple of notes here: for "material", baseball bats can be made of either wood or aluminum. It is OK for keys to be duplicated within a file. You may have a separate file of vendors, with their addresses, phone numbers, local distributors, and credit terms. Obviously, the "name" field in this file would be an index. If an exact match exists between the "name" in the "SportsRecord" file and a "name" in the "Vendor" file, you have a "relation". Instead of having a 30 character name in the "SportsRecord", you could have the record number for the entry in the "Vendor" file.

With dBC III and other compatible products, the index exists as a separate file. This file contains keys and record numbers, organized into a B-tree structure, but this is normally transparent to the user. Before a dBC III file is processed, it must be opened with a function named dBopen, and all index files associated with it must be opened with dBiopen. Both of these functions initialize a file descriptor which will be used for all future references to these files.

Now, to read a "baseball bat" record: Assume that "sportfd" is a pointer to the file descriptor for the "SportsRecord" file, and "sportitemfd" is a pointer to the file descriptor for the "sport+item" index. The following piece of code will attempt to read this record.

```
struct SportItemStruc
     {
   sport[15];
   item[17];
     } SportItemKey;

strncpy(SportItemKey.sport,"baseball           ",
    len(SportItemKey.sport));
strncpy(SportItemKey.item,"bat                  ",
   len(SportItemKey.item));
if (dBgetrk(sportfd,sportitemfd,&SportItemKey,
   &SportsRecord,&status) != SUCCESS)
   {
   printf("didn't exist\n");
     }
else
     {
   /* continue processing */
     }
```

If there actually is at least one record in the file with "baseball" in the sport field and "bat" in the item field, the call to dBgetrk should retrieve it, and copy the data into the "SportsRecord" structure. If you want to read more "baseball bat" records, you would call dBgetnr (get next record) repeatedly, until the "item" field contains something other than "bat".

dBC III has applications beyond conventional database applications. I used it to control an interactive video system. It could be used for adventure games, and in any situation where referencing data with names is important.

A routine to add a record to this file would look something like this:

```
  if (dBputr(sportfd,agentindexfd,SportsRecord.sport,
    &SportsRecord) != SUCCESS)
      {
    printf("Error adding record\n");
    break;
      }

  if (dBakey(agentindexfd,child,recnum) != SUCCESS)
      {
    printf("dBakey failed\n");
    break;
      }

  if (dBakey(itemfd,SportsRecord.item,recnum) != SUCCESS)
      {
    printf("dBakey unsuccessful\n");
    break;
      }

  if (dBakey
    (vendorfd,SportsRecord.vendor,recnum) != SUCCESS)
      {
    printf("dBakey unsuccessful\n");
    break;
      }
```

If a file has only one index, it is necessary only to call dBputk, which will write the record to the file and update the index. If a record already exists in the file which has the same key as the record being written, that record will be overwritten. If no such record exists, a new record will be created.

If a file has more than one index, you must go to some trouble. dBakey is used to add a record which is already in a file to an index. However, you must supply it with a record number and dBrkey must be called in order to find out what the record number is.

Note that there is no index for the "sport" field. It isn't necessary, because the function dBpkey, which allows you to specify a partial key, can be used to locate all records for a certain sport using the "sport item" index.

In this example, the keys are a part of the record being written. This is the typical situation, but there are other possibilities. Suppose that one of the key fields is a person's name, and this name can contain both upper- and lower-case characters. You do not want case-sensitivity, however, when you look up a person in this file. You could do the following:

```
  for (n=0; n<strlen(keyfield); n++)
    keyfield[n] = tolower(keyfield[n]);

  if (dBputk(fd, name, keyfield, record) != SUCCESS)
      {
    printf("error adding file\n");
     ...
      }
```

Now, the keys in the index are not the same as what is actually in the file, but it doesn't matter, as long as you convert a supplied key to lower case prior to calling dBgetrk to read the record. It's possible for keys to have no resemblance whatsoever to any of the data actually in the record, as long as it makes sense for your application. It would also be possible to have multiple keys for the same record, by calling dBakey to add the additional keys.

dBASE Compatibility Issues

In the early stages of your development process, you must decide whether the files that you created need to adhere to any restrictions regarding what sort of data can be contained in these files. dBC III allows you to read and write dBASE-compatible files, but does nothing in the way of enforcement. If you are creating files which will only be read by your own programs, it is perfectly OK to store binary data in fields, or use variable length and/or delimited fields. If, however, you want your files to be read by other programs capable of processing dBASE compatible files, you must follow some rules.

What programs can process dBASE-compatible files? AmigaVision uses them for database operations, but not for scripts. Organize! is a simple but useful program for putting data into files and displaying it. dBMAN (version V) is an attempt at implementing the dBASE language on the Amiga. It can read and write files processed by dBC III, but it uses a different format for indexes. This does not have to be a big problem, as long as your files are not very large; just simply reindex files prior to using them. There are some other products, Superbase, for example, that can import and export dBASE-compatible files.

If you want your application to be dBASE-compatible, you must follow the following rules:

1) Don't store C-type null-terminated strings. Instead, fill the field out with spaces. A function, dBstrcpy, is provided to do this.

2) Numeric data fields are represented as right-justified numeric digits, and a pair of functions, dBatofld and dBfldtoa, are provided to convert numeric fields from and to null-terminated ASCII strings, which may or may not contain a decimal point. If you intend to do arithmetic in C with this sort of data, you must also call atof to convert it from ASCII to binary (sprintf to convert it in the other direction). Numeric keys, however, are represented as floating-point numbers in the index file, and dBatokey is provided to do this conversion.

3) Logical fields should contain "T" or "F".

Memo Fields

dBASE files are allowed to have "memo" fields. Memo fields allow variable-length text to be associated with a data base record. This data is stored as blocks within a separated file, and a pointer to the block is stored as a ten-digit number in the actual data base record. dBgetm and dBputm are provided to deal with memo fields.

Processing of memo fields by dBC III is painfully slow. When a memo field is written, the high-order bit of every byte in the text is turned on, for no obvious reason. When a memo field is read, these bits are turned off again. This makes processing memo fields very slow: it takes 3-5 seconds to read a memo field and display it on the screen of an Amiga 2000. The text used in my application often includes non-English characters, and fooling with the high-order bit in this situation is fatal. I ended up writing my own routines in C and Assembler to process memo fields and I found this to be reasonably simple to do.

Creation of Files

A function is supplied with dBC III to create files. You should try to avoid using it. Use Organize! or dBMAN instead. This rule also applies if you need to add a field to a file, remove a field from a file, or change the size of a field.

Reindexing

Sooner or later, your files will have to be reindexed. You would have to do this if the file was modified by dBMAN, or imported, changed, and then exported by Superbase. If a guru or power failure occurs when you are updating an index, or in that interval after a record is added (or a key is changed) and before the index is updated, the file and the index will be in disagreement.

A reindexing operation simply recreates an index file from scratch, by reading every record in the data file, and adding an index for each record. The easiest way to accomplish this is to use Organize! to do it. Unfortunately, Organize! can deal with only the simplest types of keys: keys may consist of only one field, and the key can only be what this field actually contains. (For example, Organize! wouldn't be able to deal with the case-insensitive example given earlier.) If your keys are more complex, you must write your own program to do reindexing, which could look something like this:

```
/*      The second parameter to dBicreat is the dBASE index
        expression
        the third parameter is the length of the index
        the fourth parameter is the index type
          C - character
          N - numeric          */
    if (dBicreat
          ("partnum.ndx","PARTNO+ITEM",32,'C') != SUCCESS)
      {printf("dBicreat unsuccessful\n");
       exit(TRUE);}

    if (dBicreat("item.ndx","ITEM",10,'C') != SUCCESS)
      {printf("dBicreat unsuccessful\n");
       exit(TRUE);}

    if (dBicreat("vendor.ndx","VENDOR",10,'C') != SUCCESS)
      {printf("dBicreat unsuccessful\n");
       exit(TRUE);}

    if (dBopen("export.dbf",&exportfd) != SUCCESS)
      {printf("dBopen unsuccessful\n");
       exit(TRUE);}

    if (dBiopen("partnum.ndx",&partnumfd) != SUCCESS)
      {printf("dBiopen unsuccessful\n");
       goto q1;}

    if (dBiopen("item.ndx",&itemfd) != SUCCESS)
      {printf("dBiopen unsuccessful\n");
       goto q2;}

    if (dBiopen("vendor.ndx",&vendorfd) != SUCCESS)
      {printf("dBiopen unsuccessful\n");
       goto q3;}

/*  find out how many records there are in the file  */
    dBsize(exportfd,&recno);


for (j=1; j<=recno; j++)
  {
    if (dBgetr(exportfd,j,&exportRecord,&status) != SUCCESS)
             {
              printf("dBgetr unsuccessful\n");
              goto q4;
             }

/* since we defined this index to be 32 bytes long, this index
        will consist of both the "partno" and "item" fields   */
    if (dBakey(partnumfd,exportRecord.partno,j) != SUCCESS)
             {
              printf("dBakey unsuccessful\n");
              goto q4;
             }
    if (dBakey(itemfd,exportRecord.item,j) != SUCCESS)
             {
              printf("dBakey unsuccessful\n");
              goto q4;
             }
    if (dBakey(vendorfd,exportRecord.vendor,j) != SUCCESS)
             {
              printf("dBakey unsuccessful\n");
              goto q4;
             }
  }

q4:
if (dBiclose(vendorfd) != SUCCESS)
       printf("dBiclose unsuccessful\n");
q3:
if (dBiclose(itemfd) != SUCCESS)
       printf("dBiclose unsuccessful\n");
q2:
if (dBiclose(partnumfd) != SUCCESS)
       printf("dBiclose unsuccessful\n");
q1:
if (dBclose(exportfd) != SUCCESS)
       printf("dBclose unsuccessful\n");
```

Note that the value of "status" returned by dBgetr is not checked, so any deleted records in this file will be included in the indices.

Record Deletion

The function of the "status" variable in the call to dBputr in the first example merits discussion. When dBdelete is called, the record specified is not actually removed from the file, it is only "marked" as deleted. If you later attempt to read a deleted record with dBgetr or another routine, the value "SUCCESS" will still be returned, but the value of "status" will be "INACTIVE" instead of "ACTIVE".

Both Organize! and dBMAN have the capability to "pack" a file. This actually removes deleted records from a file. If you don't have one of these products, you must write your own program to do it.

Don't forget that packing a file causes the "record numbers" (the physical position of a record with the file) to change.

Evaluation of dBC III

I encountered few bugs in the dBC III and the performance is good, as long as you don't use memo fields. The manual's examples are also very useful. One chapter is a complete tutorial, and all of the examples in the tutorial are provided on the diskette with the software. I only found a couple of minor ambiguities in the manual, and one serious bug. On page 7-25, a global variable named _dbmax is discussed. If you attempt to assign a different value to this, it must be defined as short, or you will be in lots of trouble. This problem is obviously a reflection of the fact that this software was developed for PCs.

dBC III has applications beyond conventional data base applications. I used it to control an interactive video system. It could be used for adventure games, and in any situation where referencing data with names is important.

✓

About the Author

Robert Broughton is a consultant in Vancouver, Canada. Robert's recent project was the development of an interactive video system, _Laser Atlas_, targeted towards the tourist industry. He can be reached c/o AC's TECH or via USEnet at a1040@mindlink.UUCP.

Using Intuition's Proportional Gadgets from Absoft's FORTRAN 77

by Joseph R. Pasek

Introduction

The Amiga has developed a reputation as an excellent low cost but powerful computer with many areas of application. Both the Amiga 3000 and the Amiga 2000 (or 2500) equipped with either an 68020/68881 or 68030/68882 board and the appropriate software application software can also become a very powerful scientific and engineering workstation. The programming language that is most often employed in the scientific and engineering applications is the venerable FORTRAN language. Due to the sheer bulk of available source code written in FORTRAN and the relative ease at which scientists and engineers program with it, it is still the language of choice.

The Amiga programmer, using Absoft's FORTRAN 77, is capable of taking advantage of most of the Amiga's ROM Kernel routines. Include files for the Amiga's graphic, intuition, layers, diskfont, dos and exec functions are provided. However, the FORTRAN's access to the Amiga's ROM Kernel routines is somewhat incomplete; for example Absoft's include files for access to Intuitions routine has no provision for the structures needed to control and use Intuition's gadgets. The Gadgets are the graphical buttons (boolean), sliders (proportional) and string objects that a program could employ to obtain user's inputs.

This FORTRAN implementation does not directly access the ROM Kernel Routines. Instead a routine is provided that is capable of interfacing to the Amiga's ROM Kernel routines. This routine is referred to as just amiga( ). Usage of the amiga( ) subroutine is as follows, as a subroutine:

```
        call amiga( ROM Function name, arg1, arg2,
..., arg n)
```

or, as a function

```
        result = amiga (ROM Function name, arg1,
arg2, ..., arg n).
```

An identical approach is employed by Absoft's Macintosh version of the FORTRAN compiler where the use of an interfacing routine called Toolbox( ) is employed (I am not sure if this still holds in the latest release of Absoft's MacFORTRAN 77).

Absoft provides examples that shows some of its ROM Kernel interfacing capability with coded examples that generate some graphics by calling the ROM Kernel using the amiga( ) interface routine.

The Languages Implementation

The Absoft's FORTRAN language compiler for the Amiga has been available for just about as long as the Amiga has been available. The most currently available version, Version 2.3, is functionally equivalent to the ANSI standard FORTRAN 77 with some additional extensions that will be found in the proposed FORTRAN 90 standard.

FORTRAN is a high level applications programming language, especially suited for the scientific and engineering areas. Contrast this with another often used Amiga language C which is considered to be primarily a systems programming language. Users of the C language are insistent that it should be considered as an all purpose language, insisting that it is capable of being applied as a high level scientific and engineering applications language. The authors of the book Numerical Recipes in C, Cambridge University Press, state in the preface that the C language does not easily lend itself to numerical analysis applications. In my opinion probably the language that best qualifies as the best all purpose language on the Amiga for both systems and applications programming would be Modula 2, but that is the subject for another time. For some of us who work in the scientific and engineering application programming areas FORTRAN remains the language of choice.

The purposes of this article are several, first to show the Amiga programming community that Absoft's FORTRAN on the Amiga is capable of providing access, although indirectly, to the Amiga's advanced features as embodied in the ROM Kernel routines, again without resorting to writing extra C or assembly language code. Second, to extend the scope of the include files that is over and above even what the compiler's developers ever contemplated being done in terms of interfacing with the Amiga ROM Kernel routines. Third, it is also hoped to educate the programming community that structured programs can come from this version of FORTRAN, since a number of the academically trained programmers while in school where subject to a great deal of anti-FORTRAN propaganda (in some cases justified) and as a result are severely biased against this language.

Most of the "bad" elements about FORTRAN were primarily based on the older FORTRAN IV. Although most elements of FORTRAN IV are still found in FORTRAN 77 (for compatibility reasons), the current version provides a large number of extensions that permit the programmer to employ more acceptable programming methodology, such as those language structures to minimize the need for the use of the dreaded GOTO's. It even provides a mechanization for strong data typing familiar to all who program in the modern high level programming languages.

OH NO! No support for the C structures

FORTRAN 77 does not support anything that resembles C's structures or Pascal's RECORDS (although FORTRAN 90 does). But there is a work-around. It can be observed that a C's data structure is nothing more than an array of data with mixed data types associated to various portions of that array. In the provided FORTRAN's include files byte arrays and equivalence statements are used together to define something that is functionally similar to C's data structures.

Figure 1 shows a portion of the Absoft's "graphic.inc" include file. The left side defines the data type of a variable name (note the variable names used are similar to the ones one would find in the Amiga's C language include files). On the right the FORTRAN's EQUIVALENCE statement is used to place the just typed variable name at the appropriate location in a defined byte array. This process is repeated until all the elements of a typical Amiga C structures are mapped into the FORTRAN byte array.

In addition to the generation of the FORTRAN equivalences of C's structures, Absoft's include files also provide defines of commonly used Amiga parameters, symbolic names, and information needed to access the ROM Kernel routines by the FORTRAN's amiga( ) interface routine. Some of the defines for ROM Kernel routines are shown in Figure 2.

Several FORTRAN include files are provided by Absoft to allow the user to interface the FORTRAN to the ROM Kernel routines. These are:

```
graph.inc
exec.inc
intuit.inc
layers.inc
dskfnt.inc
dos.inc
```

As mentioned Absoft does provide some demo programs to show how to use the information in the include files with the FORTRAN to interface with the ROM Kernel routines.

The Original Effort

The examples presented here arose from work being done in rehosting signal processing software written in FORTRAN that originally resided on a DEC's VAX computer. Initially, the port was done in a straight forward manner with little effort made to take advantage of any of the Amiga's interface features outside of the CLI.

The program was completely rehosted and tested. However, further work was pre-empted by a change in work assignment. Upon return to the this effort being a bit wiser, and upon review of the software and its potential user base it was decided that working from the CLI was adequate, but this ran contrary to the Amiga's graphical interface philosophy. The Amiga with its graphical user interface lead me to the conclusion there was no reason why this software port should be tied to the CLI just because it was the easiest to implement.

The challenge was to install in the FORTRAN source code the capability to provide the user of that FORTRAN based application program some user friendly interface elements. From the nature of the inputs needed by the program, the form of the desired means of input was identified. The identification process required the use of various gadget types string, boolean and proportional. The use of such gadgets requires the programmer to employ one or more of the following structures:

```
StringInfo
PropInfo
Gadget
```

In addition to the structures, the use of the appropriate ROM Kernel routines are needed to either set up or control the gadgets.

Examination of Absoft's FORTRAN's intuition include files (intuit.inc) showed that the needed gadget structures were not available. However, the information needed to access the ROM Kernel routines in the same include file appeared to be almost all there. Figure 3 has the modifications or additions needed in order for the FORTRAN to have control of the gadgets.

The specific code segments needed to define the Gadget structures in Absoft's intuit.inc files are shown in Figure 4.

Usage of the include files also showed a couple of the references needed to access some of the ROM Kernel were either absent or wrong. The intuit.inc file's reference to the ROM Kernel call EndRequest was in error. It must be changed from z'00385314' to z'00328314'. The second find was that there was no reference to the ActivateGadget in the intuit.inc include file. The following line must be also added to the intuit.inc file

```
integer ActivateGadget  ;  parameter
(ActivateGadget =z'06H25340')
```

With this last change to the FORTRAN's intuit.inc file it is now possible to write some code in FORTRAN that permits the user to use the Amiga's gadgets.

Figure One

A small portion of AC/FORTRAN's graphics include file is shown here. In particular, the AreaInfo and TextAttr structures are defined here. Also shown are parameters that are typed and assigned preset values.

```
integer*2 AreaInfo(12)
integer*4 ai_VctrTbl                 ; equivalence (AreaInfo(1), ai_VctrTbl)
integer*4 ai_VctrPtr                 ; equivalence (AreaInfo(3), ai_VctrPtr)
integer*4 ai_FlagTbl                 ; equivalence (AreaInfo(5), ai_FlagTbl)
integer*4 ai_FlagPtr                 ; equivalence (AreaInfo(7), ai_FlagPtr)
integer*2 ai_Count                   ; equivalence (AreaInfo(9), ai_Count)
integer*2 ai_MaxCount                ; equivalence (AreaInfo(10),ai_MaxCount)
integer*2 ai_FirstX                  ; equivalence (AreaInfo(11),ai_FirstX)
integer*2 ai_FirstY                  ; equivalence (AreaInfo(12),ai_FirstY)

* rp_Flags:

integer FRST_DOT                     ; parameter (FRST_DOT = z'0001')
integer ONE_DOT                      ; parameter (ONE_DOT = z'0002')
integer DBUFFER                      ; parameter (DBUFFER = z'0004')
integer AREAOUTLINE                  ; parameter (AREAOUTLINE = z'0008')
integer NOCROSSFILL                  ; parameter (NOCROSSFILL = z'0020')

* rp_DrawMode:

integer JAM1                         ; parameter (JAM1 = 0)
integer JAM2                         ; parameter (JAM2 = 1)
integer COMPLEMENT                   ; parameter (COMPLEMENT = 2)
integer INVERSVID                    ; parameter (INVERSVID = 4)

* rp_TxFlags:

integer TXSCALE                      ; parameter (TXSCALE = 1)

* - from "text.i" -
integer*1 TextAttr(8)

integer*4 ta_Name                    ; equivalence (TextAttr(1),ta_Name)
integer*2 ta_YSize                   ; equivalence (TextAttr(5),ta_YSize)
integer*1 ta_Style                   ; equivalence (TextAttr(7),ta_Style)
integer*1 ta_Flags                   ; equivalence (TextAttr(8),ta_Flags)

* ta_Style:

integer FS_NORMAL                    ; parameter (FS_NORMAL = 0)
integer FS_UNDERLINED                ; parameter (FS_UNDERLINED = 1)
integer FS_BOLD                      ; parameter (FS_BOLD = 2)
integer FS_ITALIC                    ; parameter (FS_ITALIC = 4)
integer FS_EXTENDED                  ; parameter (FS_EXTENDED = 8)

* ta_Flags:

integer FP_ROMFONT                   ; parameter (FP_ROMFONT=   1)
integer FP_DISKFONT                  ; parameter (FP_DISKFONT=   2)
integer FP_REVPATH                   ; parameter (FP_REVPATH     4)
integer FP_TALLDOT                   ; parameter (FP_TALLDOT=    8)
integer FP_WIDEDOT                   ; parameter (FP_WIDEDOT=  16)
integer FP_PROPORTIONAL              ; parameter (FP_PROPORTIONAL =  32)
integer FP_DESIGNED                  ; parameter (FP_DESIGNED =  64)
integer FP_REMOVED                   ; parameter (FP_REMOVED= 128)
```

Figure Two

This shows another portion of the FORTRAN graphics.inc file; in this case the ROM Kernel function names are listed and assigned numeric values. The value assigned each function name is utilized by the Amiga() routine to interface to the corresponding ROM Kernel routines.

```
function values for 'graphics.library'

integer GfxBase                  ; parameter (GfxBase          =z'00000200')
integer AndRegionRegion          ; parameter (AndRegionRegion  =z'00338268')
integer XorRegionRegion          ; parameter (XorRegionRegion  =z'00338267')
integer OrRegionRegion           ; parameter (OrRegionRegion   =z'00338266')
integer BltBitMapRastPort        ; parameter (BltBitMapRastPort=z'000FF265')
integer FreeGBuffers             ; parameter (FreeGBuffers     =z'14328264')
integer CopperListInit           ; parameter (CopperListInit   =z'00208263')
integer ScrollVPort              ; parameter (ScrollVPort      =z'00008262')
integer GetRGB4                  ; parameter (GetRGB4          =z'00218261')
integer FreeColorMap             ; parameter (FreeColorMap     =z'00018260')
integer GetColorMap              ; parameter (GetColorMap      =z'0001025F')
integer FreeCprList              ; parameter (FreeCprList      =z'0001825E')
integer XorRectRegion            ; parameter (XorRectRegion    =z'0013825D')
integer ClipBlit                 ; parameter (ClipBlit         =z'000FF25C')
integer FreeCopList              ; parameter (FreeCopList      =z'0001825B')
integer FreeVPortCopLists        ; parameter (FreeVPortCopLists=z'0001825A')
integer DisposeRegion            ; parameter (DisposeRegion    =z'00018259')
integer ClearRegion              ; parameter (ClearRegion      =z'00018258')
integer NotRegion                ; parameter (NotRegion        =z'00018257')
integer NewRegion                ; parameter (NewRegion        =z'00000256')
integer OrRectRegion             ; parameter (OrRectRegion     =z'00338255')
integer AndRectRegion            ; parameter (AndRectRegion    =z'00338254')
integer FreeRaster               ; parameter (FreeRaster       =z'04018253')
```

Figure Three

Modifications and additions made to the FORTRAN's intuition File.

Added the following Intuition Structures to Absoft's intuition include file.

```
        StringInfo
        PropInfo
        Gadget
```

Modified the information needed to access the Intuition's EndRequest routine (See Figure 2).

Added the routine ActivateGadget function to the Intuition's include file function list.

Figure Four

This is a listing of the FORTRAN templates (equivalent to C's structures) and parameter definitions that must be added to Absoft's FORTRAN include file. Once these code segments are added, the Amiga FORTRAN user is capable of accessing Intuition's string, boolean, and proportional gadgets.

StringInfo structure

```
integer*1 StringInfo(36)

integer*4 si_Buffer                    ; equivalence (StringInfo(1),si_Buffer)
integer*4 si_UndoBuffer                ; equivalence (StringInfo(5),si_UndoBuffer)
integer*2 si_BufferPos                 ; equivalence (StringInfo(9),si_BufferPos)
integer*2 si_MaxChars                  ; equivalence (StringInfo(11),si_MaxChars)
integer*2 si_DispPos                   ; equivalence (StringInfo(13),si_DispPos)
integer*2 si_UndoPos                   ; equivalence (StringInfo(15),si_UndoPos)
integer*2 si_NumChars                  ; equivalence (StringInfo(17),si_NumChars)
integer*2 si_DispCount                 ; equivalence (StringInfo(19),si_DispCount)
integer*2 si_CLeft                     ; equivalence (StringInfo(21),si_CLeft)
integer*2 si_CTop                      ; equivalence (StringInfo(23),si_CTop)
integer*4 si_LayerPtr                  ; equivalence (StringInfo(25),si_LayerPtr)
integer*4 si_LongInt                   ; equivalence (StringInfo(29),si_LongInt)
integer*4 si_AltKeyMap                 ; equivalence (StringInfo(33),si_AltKeyMap)
```

Gadget structure

```
integer*1 Gadget(44)

integer*4 gg_NextGadget                ; equivalence (Gadget(1), gg_NextGadget)
integer*2 gg_LeftEdge                  ; equivalence (Gadget(5), gg_LeftEdge)
integer*2 gg_TopEdge                   ; equivalence (Gadget(7), gg_TopEdge)
integer*2 gg_Width                     ; equivalence (Gadget(9), gg_Width)
integer*2 gg_Height                    ; equivalence (Gadget(11), gg_Height)
integer*2 gg_Flags                     ; equivalence (Gadget(13), gg_Flags)
integer*2 gg_Activation                ; equivalence (Gadget(15), gg_Activation)
integer*2 gg_GadgetType                ; equivalence (Gadget(17), gg_GadgetType)
integer*4 gg_GadgetRender              ; equivalence (Gadget(19), gg_GadgetRender)
integer*4 gg_SelectRender              ; equivalence (Gadget(23), gg_SelectRender)
integer*4 gg_GadgetText                ; equivalence (Gadget(27), gg_GadgetText)
integer*4 gg_MutualExclude             ; equivalence (Gadget(31), gg_MutualExclude)
integer*4 gg_SpecialInfo               ; equivalence (Gadget(35), gg_SpecialInfo)
integer*2 gg_GadgetID                  ; equivalence (Gadget(39), gg_GadgetID)
integer*4 gg_UserData                  ; equivalence (Gadget(41), gg_UserData)
```

Gadget flags

```
integer GADGHIGHBITS                 ; parameter (GADGHIGHBITS = z'0003')
integer GADGHCOMP                    ; parameter (GADGHCOMP    = z'0000')
integer GADGHBOX                     ; parameter (GADGHBOX     = z'0001')
integer GADGHIMAGE                   ; parameter (GADGHIMAGE   = z'0002')
integer GADGHNONE                    ; parameter (GADGHNONE    = z'0003')
integer GADGIMAGE                    ; parameter (GADGIMAGE    = z'0004')
integer GRELBOTTOM                   ; parameter (GRELBOTTOM   = z'0008')
integer GRELRIGHT                    ; parameter (GRELRIGHT    = z'0010')
integer GRELWIDTH                    ; parameter (GRELWIDTH    = z'0020')
integer GRELHEIGHT                   ; parameter (GRELHEIGHT   = z'0040')
integer SELECTED                     ; parameter (SELECTED     = z'0080')
integer GADGDISABLED                 ; parameter (GADGDISABLED = z'0100')
integer RELVERIFY                    ; parameter (RELVERIFY    = z'0001')
integer GADGIMMEDIATE                ; parameter (GADGIMMEDIATE = z'0002')
integer ENDGADGET                    ; parameter (ENDGADGET    = z'0004')
integer FOLLOWMOUSE                  ; parameter (FOLLOWMOUSE  = z'0008')
integer RIGHTBORDER                  ; parameter (RIGHTBORDER  = z'0010')
integer LEFTBORDER                   ; parameter (LEFTBORDER   = z'0020')
integer TOPBORDER                    ; parameter (TOPBORDER    = z'0040')
integer BOTTOMBORDER                 ; parameter (BOTTOMBORDER = z'0080')
integer TOGGLESELECT                 ; parameter (TOGGLESELECT = z'0100')
integer STRINGCENTER                 ; parameter (STRINGCENTER = z'0200')
integer STRINGRIGHT                  ; parameter (STRINGRIGHT  = z'0400')
integer LONGINT                      ; parameter (LONGINT      = z'0800')
integer ALTKEYMAP                    ; parameter (ALTKEYMAP    = z'1000')
```

Gadget types

```
integer GADGETTYPE                   ; parameter (GADGETTYPE   = z'FC00')
integer SYSGADGET                    ; parameter (SYSGADGET    = z'8000')
integer SCRGADGET                    ; parameter (SCRGADGET    = z'4000')
integer GZZGADGET                    ; parameter (GZZGADGET    = z'2000')
integer REQGADGET                    ; parameter (REQGADGET    = z'1000')
integer SIZING                       ; parameter (SIZING       = z'0010')
integer WDRAGGING                    ; parameter (WDRAGGING    = z'0020')
integer SDRAGGING                    ; parameter (SDRAGGING    = z'0030')
integer WUPFRONT                     ; parameter (WUPFRONT     = z'0040')
integer SUPFRONT                     ; parameter (SUPFRONT     = z'0050')
integer WDOWNBACK                    ; parameter (WDOWNBACK    = z'0060')
integer SDOWNBACK                    ; parameter (SDOWNBACK    = z'0070')
integer CLOSE                        ; parameter (CLOSE        = z'0080')
integer BOOLGADGET                   ; parameter (BOOLGADGET   = z'0001')
integer GADGET0002                   ; parameter (GADGET0002   = z'0002')
integer PROPGADGET                   ; parameter (PROPGADGET   = z'0003')
integer STRGADGET                    ; parameter (STRGADGET    = z'0004')
```

PropInfo structure

```
integer PropInfo(22)

integer*2 pi_Flags                    ; equivalence (PropInfo(1), pi_Flags)
integer*2 pi_HorizPot                 ; equivalence (PropInfo(3), pi_HorizPot)
integer*2 pi_VertPot                  ; equivalence (PropInfo(5), pi_VertPot)
integer*2 pi_HorizBody                ; equivalence (PropInfo(7), pi_HorizBody)
integer*2 pi_VertBody                 ; equivalence (PropInfo(9), pi_VertBody)
integer*2 pi_CWidth                   ; equivalence (PropInfo(11), pi_CWidth)
integer*2 pi_CHeight                  ; equivalence (PropInfo(13), pi_CHeight)
integer*2 pi_HPotRes                  ; equivalence (PropInfo(15), pi_HPotRes)
integer*2 pi_VPotRes                  ; equivalence (PropInfo(17), pi_VPotRes)
integer*2 pi_LeftBorder               ; equivalence (PropInfo(19), pi_LeftBorder)
integer*2 pi_TopBorder                ; equivalence (PropInfo(21), pi_TopBorder)
```

PropInfo flags

```
 integer AUTOKNOB                     ; parameter (AUTOKNOB         = z'0001')
 integer FREEHORIZ                    ; parameter (FREEHORIZ        = z'0002')
 integer FREEVERT                     ; parameter (FREEVERT         = z'0004')
 integer PROPBORDERLESS               ; parameter (PROPBORDERLESS   = z'0008')
 integer KNOBHIT                      ; parameter (KNOBHIT          = z'0100')
 integer KNOBHMIN                     ; parameter (KNOBHMIN         = z'0006')
 integer KNOBVMIN                     ; parameter (KNOBVMIN         = z'0004')
 integer MAXBODY                      ; parameter (MAXBODY          = z'ffff')
 integer MAXPOT                       ; parameter (MAXPOT           = z'ffff')
```

An Example: A Proportional Gadget from FORTRAN code (On Disk!)

DriveTime is simple. It sets up the Amiga's NewScreen structure, and then proceeds to call OpenScreen using the amiga( ) routine. It is from this newly opened screen that the window containing the proportional gadget is found. The proportional gadget is set up and used by the HamReq subroutine.

The HamReq subroutine first allocates some chip memory for the slider knob image of the proportional gadget. This is achieved with a call to the AllocMem routine and a request is made for 200 bytes of CHIP memory. The AllocMem routine returns the base address of the memory allocated. The slider's knob image as defined in the KnobGraphic array is moved into the just allocated chip memory. This step uses the Absoft's FORTRAN word( ) function to place the arrays data at the desired memory location. The next step sets up AKnobImage array (or data structure) by employing the Image template as defined in the intuition file. This is followed by setting up the APropInfo array from the PropInfo template.

This defines the proportional gadgets attributes. An IntuiText structure is next defined into the IText array. The Gadget1 array is next defined from the Gadget template again defined in the intuition file. Again the nomenclature used here is nearly identical to that used in the Amiga's C structure definitions. A window structure called FirstNewWindow is then defined using the NewWindow template. All the needed structures are now defined.

A call to OpenLibrary is done to open the Amiga's Intuition.library. As required by the operating system the strings sent to the operating system must be zero (0) terminated. A check is made to determine if the library has been opened. The NewWindow template is used to set up the FirstNewWindow structure. To the FirstNewWindow structure is attached the proportional gadget structure Gadget1 by passing the Gadget1 address pointer to FirstGadg. A call is made to the OpenWindow routine to open the window. If the first window cannot be opened the opened library is closed and the program is halted returning control to the user.

Absoft also provides several routines that are not found in the standard FORTRAN. The amiga( ) routine has already been discussed. The other provided routine is loc( ) which returns a pointer to a FORTRAN variable. Examination of the code described thus far shows some applications of the loc( ) routine. Absolute memory addressing is also supported by the FORTRAN compiler with three functions byte( ), word( ), and long( ) word memory addressing. The code description that follows will show application of these special intrinsic functions.

The windows raster port pointer (RPortPtr) is derived from the memory offset wd_RPort and the windows FirstWindow base pointer. Here the absolute memory addressing function long( ) is employed. The SetAPen function is called to set the pen color. The variable HAM_FACTOR is converted to longint which ranges from 0 to FFFF. The floating point variable HAM_FACTOR is converted to string ham_temp based on a format description. Move( ) is called to set the position of the slider object in the proportional gadget. The value of HAM_FACTOR variable is placed via the IntuiText facility into the window. The do-while loop that follows monitors the messages that are received by the window through its user's port.

A provision is made to suspend any processing until some form of input is directed to the window, this is achieved calling the Amiga's Wait routine. The Wait routine suspends this process until it is determined that some input has been directed to this window, this allows the FORTRAN based process to be more cooperative in Amiga's multi tasking environment. When some mouse activity is detected that is in accord with the windows pre-defined IDCMPFlags the process is activated and the type of message (MessageClass) is ascertained.

For this example only two message classes are looked for: GADGETUP and CLOSEWINDOW. The GADGETUP message indicates that the window's proportional gadget has been manipulated. The gadget's positional change is taken from the APropInfo structure converted to a string, and displayed as some IntuiText. A CLOSEWINDOW messageclass is activated by clicking on the window's close gadget. Upon detection the RemoveGadget system routine is called to delete the gadget from the window. Followed by a call to the Amiga systems CloseWindow routines to close the window and finally the intuition.library is closed for this process. The do-while loop variable Wstat is changed to a false condition allowing the loop to be exited.

The final steps entail setting the HAM_FACTOR to the most recent values as ascertained from the last proportional gadget setting. The bit of chip memory allocated by a call AllocMem is de-allocated by call to the system's FreeMem routine.

The executable form of the program described is called TestProp. Access from the workbench is achieved by clicking the TestProp_Demo icon. The Source code is provided in Prop_Source directory. Additional examples of FORTRAN based code are also provided, click on the Fills_Demo and Jupiter_Moons icons. The Jupiter Moons is a Jupiter Moons simulation written in FORTRAN. Fills_Demo is an example of how to access the Amiga's areafill routines.

There, in a relatively large nutshell, is an example of how to interface one's FORTRAN code to the Amiga's Intuition to take advantage of its proportional gadgets. Future articles will describe the interfacing of FORTRAN source to other Intuition gadgets, string and boolean. ☑

FastBoot
A Super BootBlock

Creating a bootable, recoverable, RAM disk

by Dan Babcock

What is it?

FastBoot is a bootblock that quickly loads an entire disk into memory, creates a RAM disk and boots from that RAM disk. The RAM disk that FastBoot creates is recoverable and autobooting (it requires Kickstart 1.3 or later). It is equivalent to mounting a floppy-sized RAD: (more properly known as ramdrive.device), using DiskCopy to fill RAD:, ejecting the floppy, and resetting the machine. Because FastBoot resides solely on the bootblock, however, all these functions require no usable disk space—and proceed as quickly as possible (no endless disk grinding!). FastBoot has other advantages too: it supports two popular Amiga 1000 hacks (512K of piggyback RAM at $80,000 and Kickstart-in-EPROM with 256K of RAM at $F80,000) and it allocates its memory in four 220K chunks (RAD:, by contrast, allocates a single contiguous 880K chunk); allocating in four chunks permits contiguous memory regions as small as 256K to be utilized (such as results from the Kickstart-in-EPROM hack). FastBoot is also flexible; it may be bypassed completely by pressing the left mouse button (port 1) during boot up and, once installed, may be deinstalled by pressing the "fire" button of a joystick or mouse in port 2. Finally, FastBoot is convenient: there is no need to add files to the disk and edit startup-sequences and mountlists, making it particularly handy for speeding up games or demos with heavy disk access (assuming they don't access the floppy hardware directly).

How to use it

The source code listing was designed to be assembled with Macro68. If you lack Macro68 you'll find the binary on the accompanying disk. Once you have the 1024 byte file in hand, all that remains to be done is to install it in the bootblock of your disk(s). For this task I recommend DBInstall, a bootblock manipulator written by Dr. Bit, a Danish assembly language programmer. Again, DBInstall is found on the disk that comes with AC's Tech. To install FastBoot on a disk in drive df0:, type:

```
[illegible]
```

There are just a few points to keep in mind when using FastBoot. First of all, it requires at least 1.25MB of memory to be useful. In fact, FastBoot checks how much memory is available; if 1MB (note that this value may be easily changed in the source code) or less is available, the disk will boot normally, bypassing the special FastBoot code. If that memory requirement is too much for your system, FastBoot may be modified rather easily to support 440K RAM disks. Another requirement is Kickstart 1.3 or later. If an older Kickstart is in use, FastBoot refuses to do its magic, and the disk boots normally.

As mentioned earlier, FastBoot may be bypassed by pressing the left mouse button during boot up. Once the disk is loaded into memory, the RAM disk that was created will become the boot disk. To refer to the RAM disk, use the designation "WB:" (this is analogous to names such as "DF0:" and "DF1:"). The device name (analogous to "trackdisk.device") is also "WB".

The contents of the RAM disk are preserved when the system is reset and, in addition, AmigaDOS will boot from the RAM disk after a keyboard reset if there is no floppy in DF0:. Only one FastBoot RAM disk in memory at one time is supported. If FastBoot detects that it has been run before, a disk with the FastBoot bootblock will boot normally rather than be loaded into memory. To kick out the FastBoot RAM disk (and also RAD: if it is in memory), hold down the "fire" button of a mouse or joystick in port 2 after resetting. Port 1 was not used for this purpose to avoid a conflict with the boot selection screen in Kickstart 2.0.

Technical details

Perhaps the most interesting aspect of FastBoot is that it (like ramdrive.device on the 1.3 Workbench) is recoverable and bootable. This process revolves around two pointers present in ExecBase since version 1.2: KickMemPtr (ExecBase+$222) and KickTagPtr (ExecBase+$226).

KickMemPtr points to a list of MemLists. During startup, Exec tries to allocate the memory defined in the MemLists with AllocAbs, which attempts to mark a region of memory at an absolute address as unavailable. It succeeds if the memory has not already been allocated. A recoverable RAM disk can use KickMemPtr to ensure that its memory is not stomped on after a reset. There is one catch, though: at the time Exec calls AllocAbs, memory expansion boards have not been configured, and Exec is unaware of them. The only memory known to Exec at this time is Chip memory and the special $C00,000 memory, which means, essentially, that only Chip memory may be allocated using KickMemPtr.

KickTagPtr points to a table with the following format:

```
pointer to a romtag
pointer to a romtag
...
0 (end of table) or another KickTagPtr, identified by setting
the MSB (most significant bit)
```

A romtag is simply a table that both identifies and describes a device driver or library (often simply referred to as a module). If the AllocAbs calls made previously (when processing KickMemPtr) succeeded, then Exec gathers these romtags in a list, along with the real romtags (those that actually exist in ROM). Note that they are sorted according to the priority field of the romtags. After that, Exec calls InitCode, which calls InitResident for each module, which, among other things, calls an initialization routine in the module. Note that there is a checksum of the lists/tables associated with these two pointers. When a program alters this information, it must call the Exec routine SumKickData and store the result (in d0.l) in KickCheckSum (ExecBase+$22A).

The FastBoot code resides in Chip RAM and uses the KickMemPtr mechanism to protect itself from being overwritten. The actual RAM disk can't be expected to fit into chip RAM, however. The solution: The RAM disk data is recovered (by calling AllocAbs) in the initialization routine of the driver. Because we've chosen a priority for the driver lower than the priority of the expansion.library, RAM cards have already been configured and all memory is available. It turns out that this procedure is a bit trickier than it first appears, however. The problem is that we are making multiple calls to AllocAbs. Each AllocAbs call may consume memory if the memory list needs to be expanded but, since the other memory blocks are as yet unknown to the OS, they may be overwritten. Fortunately, there is a simple solution: Provide an eight-byte buffer area on both sides of a memory chunk.

Once the RAM disk memory has been recovered, FastBoot needs to inform the OS that it wants to autoboot. This is accomplished by enqueueing a BootNode structure on the eb_MountList of the expansion.library. To make a long story short, the following code and structures are all you need to know in order to autoboot.

Typical autoboot code

Please note that the following code assumes that the driver (called "backdisk.device" in this example) has already been initialized. Unless this happens automatically for some reason (for example, if it is hooked into KickTagPtr) you should call InitResident first. This code uses the new 68000 syntax developed by Motorola to support the added instructions and addressing modes of the 68020 and above.

```
        movea.l  (4).w,a6
        lea      (expansion,pc),a1
        moveq    #33,d0     ;enable on kickstart 1.3 or later
        SYS      OpenLibrary
        tst.l    d0
        beq.s    WrongVersion
        movea.l  d0,a5
        lea      (parmpacket,pc),a0
        SYS      MakeDosNode
        lea      (eb_MountList,a5),a0          ;address of
                                               ;eb_MountList
        movea.l  (4).w,a6
        lea      (BootNode,pc),a1
        move.l   d0,(16,a1)          ;store DosNode pointer in
                                     ;BootNode
                                     ;structure
        SYS      Enqueue             ;add BootNode to
                                     ;eb_MountList
WrongVersion:
        rts
expansion: dc.b 'expansion.library',0
        even
```

Autoboot structures

```
BootNode:
        dc.l     0          ;linkage pointer - filled in by
                            ;Enqueue
        dc.l     0          ;linkage pointer - filled in by
                            ;Enqueue
        dc.b     16         ;constant - must be here
        dc.b     priority
        dc.l     ConfigDev
        dc.w     0          ;flags - not too important
        dc.l     DosNode    ;offset 16 - filled in by above code

ConfigDev:
        dcb.b    16,0       ;ignored
        dc.b     16         ;constant - must be here
        dcb.b    15,0       ;ignored
        dc.l     DiagArea

DiagArea:
        dc.b     16         ;constant - must be here
        dcb.b    5,0        ;ignored
        dc.w     BootPoint-DiagArea          ;This is UNSIGNED.
The BootPoint
                            ;code MUST lie after the DiagArea,
                            ;not before.

BootPoint:
;standard boot code
        movea.l  (4).w,a6
        lea      (DosName,pc),a1
        SYS      FindResident
        movea.l  d0,a0
        movea.l  (22,a0),a0
        jmp      (a0)
DosName: dc.b 'dos.library',0
        even

parmpacket:
;values for a floppy-like device are given as an example
        dc.l     DevName
        dc.l     ExecName
        dc.l     0          ;unit number (not very important)
        dc.l     0          ;OpenDevice flags (not used in this
                            ;case)
        dc.l     11         ;upper bound of this table, in
                            ;longwords
        dc.l     128        ;number of longwords in a block
(512/4)
        dc.l     0          ;sector origin (never used)
        dc.l     2          ;number of surfaces
        dc.l     1          ;sectors per logical block (allocation
                            ;size) - never
                            ;used
        dc.l     11         ;sectors per track
        dc.l     2          ;reserved blocks - 2 boot blocks
        dc.l     0          ;never used
        dc.l     0          ;interleave
        dc.l     0          ;lower cylinder
        dc.l     79         ;upper cylinder
        dc.l     5          ;number of buffers
DevName: dc.b 'RAM0',0
        even
ExecName: dc.b 'backdisk.device',0
        even
```

The RAM disk driver

One of the most remarkable aspects of FastBoot is that it contains a complete device driver embedded in it—and, in fact, the device driver constitutes a rather small percentage of FastBoot's 1,006 bytes. Due to the size and complexity of the sample device driver found in the *Amiga ROM Kernel Reference Manual*, one might believe that device drivers are difficult to understand and write. Examining the device driver in FastBoot is an excellent way to learn the basic form and function of a device driver under Exec without being bogged down with unnecessary detail. The reason why the sample driver in the *ROM Kernel Reference Manual* is so complicated is that it permits reentrant operation (which is required by Exec) despite controlling nonsharable hardware (although the sample driver functions as a RAM disk, and thus doesn't need that capability).

The RAM disk that FastBoot creates is recoverable and autobooting (it requires Kickstart 1.3 or later).

That's confusing, so I'll try to clarify. Consider two tasks that both attempt to read from a floppy drive. One of the tasks calls DoIO, which causes Exec to call the driver's BeginIO entry point. The drive dispatches the request to its read routine, and starts programming the floppy controller registers. At the same time, the other task issues a read request. Once again, the driver starts programming the floppy controller registers. Chaos ensues because the other task is doing the same thing. It's clear that the floppy driver must queue and arbitrate these requests. Hence, the added complexity.

None of that is required for a RAM disk. Any number of accesses to the RAM disk may occur simultaneously with no need for queuing requests. Admittedly, there is one possible catch. If one task reads a sector at the same time another task is writing a sector, there may be confusion. However, proper file system design should not permit such a case to arise.

Miscellaneous Notes

There are a few other aspects of FastBoot worthy of commentary. For one, note that FastBoot includes a built-in Exec bug fix—equivalent to the "F" option of SetPatch—that is called via the ColdCapture vector. Since we're on the topic of Exec "capture" vectors, I'll like to digress and describe them. Because of their popular use in viruses, it's nice to know what they are.

ColdCapture: ExecBase+42. This vector is called (with a JMP) very early in the bootup routine. The return address is in A5.

CoolCapture: ExecBase+46. This vector is called (with a JSR) just before InitCode is called.

WarmCapture: ExecBase+50. This vector is called (with a JSR) right after InitCode is called.

Another notable point is that there is a quirk in the autoboot process. The floppy drive has a priority of five. If a device has a priority lower than five (please note that I'm referring to the priority field of the BootNode, not of the device driver per se), the floppy will be checked first, then (if a floppy is not present) AmigaDOS will boot from the next highest priority autoboot device. It is sometimes desirable to bypass the floppy check entirely. Take FastBoot, for instance. After FastBoot loads the floppy into memory, we want AmigaDOS to boot from the copy in RAM, not the disk. To accomplish that, one is supposed to specify a priority greater than five. Unfortunately (here's the quirk I promised you) it doesn't work quite right. It does indeed bypass the floppy check, but then there are two DF0: devices in the device list...clearly an unstable situation. FastBoot includes a hack that fixes this feature/bug. Strangely enough, the same technique does not work at all under 2.0, so FastBoot takes a different approach in that case (resetting the machine).

One other point worth noting is that AmigaDOS crashes if two volumes (i.e., disks) have the same name and creation date. Accordingly, it is not sufficient to simply copy a disk into memory verbatim—either the name or date must be changed. By far the simplest method of handling this requirement is to increment the "tick" value. A tick is, according to the *AmigaDOS Technical Reference Manual*, a fiftieth of a second. It's too small for most to be concerned with, but it is still quite sufficient for AmigaDOS to use to distinguish between two volumes with the same name. An added benefit of changing the creation time (as opposed to the volume name, for instance) is that the new checksum (It seems like everything has a checksum, doesn't it?) is easy to calculate: just subtract one.

The End...for now

By far the hardest part of writing FastBoot was dealing with the extremely limited space available—just 1012 bytes (though the current version of FastBoot only uses 1006 bytes). When confronted with such limitations, it's tempting to take shortcuts. For example, code size could have been reduced considerably by relying on certain absolute addresses in the Kickstart 1.3 ROM, but I wanted the program to be compatible with Kickstart 2.0 and beyond. I had to rewrite the code many times to pack it into the limited space. I'm sure you'll find many of your favorite optimizations in the source code listing. High-quality assembly language programming is not obsolete!

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
Absoft	15	166
Benetech Electronic Supply	47	128
Black Belt Systems	31	118
Boone Technologies	67	194
Creative Focus	81	132
Delphi Noetic	61	199
Delta Research	43	197
DKB Software	76	159
GT Devices	22	109
ICD	CIII	125
Interactive Video Systems	CII	140
M.A.S.T.	3	160
Meggido Enterprises	69	144
Memory Location,The	79	107
Memory Location,The	69	156
Prospect Technics Inc	CIV	151
Puzzle Factory,The	5	168
SAS Institute Inc	53	146
The Krueger Company	14	116
Video/ Visual Media Tools	42	157
Virtual Reality Labs	80	131

```
;*************************************************************
; FastBoot: A Super Boot Block!
;
; Written by Dan Babcock, January 1990
; Copyright (C) 1990 by Dan Babcock
;
; It's OK to copy this as long as no money is involved.
; (Commercial developers should contact me.)
;
;I may be reached on People/Link (my ID is DANBABCOCK) or Internet
;(    [illegible]).
;*************************************************************

         exeq     ;make raw code
         objfile  'df0:boot'
         super    ;suppress warnings about supervisor mode

;This source was assembled with Macro68, the best assembler ever.

;--------     User-settable parameters    ----
MemoryRequirement equ     $10000
BootPriority      equ     0         ;ranges from -128 to 127
;

RamPtr   equ      1096 ;this refers to the area of memory 1096
                  ;bytes after the bootblock start that holds
                  ;pointers to the 4 220K memory blocks

;These four addresses are used to refer to the four
;memory block pointers (logically).
mem1     equ      RamPtr
mem2     equ      RamPtr+4
mem3     equ      RamPtr+8
mem4     equ      RamPtr+12

KickMemPtrData    equ     1072 ;this doesn't refer to KickMemPtr per
                          ;se, but to the data that KickMemPtr
                          ;will point to

;Start of main code
FastBoot:
;* Read in all 880K of the disk *
         movem.l  d0-d7/a0-a6,-(sp) ;save all registers
         move.l   a1,-(sp) ;save IO request block pointer for future
                          ;use

;Check to see if FASTBOOT has already been run
         lea      (Name,pc),a1
         SYS      FindResident
         tst.l    d0
         bne      SkipNormalBoot    ;if present, boot normally

;Check Kickstart version
         bsr      CheckVersion
         bcs.b    squeeze  ;if lower than 1.2, boot normally

         bsr      addmem1  ;add A1000 memory if necessary

;Check fire button (usually left mouse button) in the left
;port. If the button is pressed, skip FASTBOOT and boot
;normally.
         btst     #6,($bfe001)
         beq.b    squeeze  ;go if pressed

;Check available memory - only perform the function if the
;user has more than MemoryRequirement bytes free. Commercial developers
;(and others) may want to customize this value to match
;the memory requirements of the program.
         moveq    #0,d1      ;no special requirements
         SYS      AvailMem
         cmp.l    #MemoryRequirement,d0        ;minimum free memory required
         bcs      SkipNormalBoot    ;go if d0.l (free mem) <
                                    ;MemoryRequirement

;Allocate 1,124 bytes of CHIP CLEAR memory and copy
;the driver (about 1,012 bytes) into it.
         move.l   #1124,d7 ;size of driver including scratch area
         move.l   d7,d0
         move.l   #MEMF_CHIP+MEMF_CLEAR,d1    ;MUST be chip RAM
         SYS      AllocMem
         movea.l  d0,a3      ;a3 is a pointer to the copy of the driver
```

```
          move.l  d0,a4
          lea     (FastBoot,pc),a0
          move.w  #Enddriver,d2
copydriver:
          move.b  (a0)+,(a4)+
          subq.w  #1,d2
          bne.b   copydriver

;Re create codeblock
          lea     (RelocTable,pc),a0 ;get pointer to relocation table
          move.l  a3,d1     ;a3 is a pointer to the copy of the driver
relocloop:
          move.w  (a0)+,d0
          beq.b   branchtorelocated       ;a zero value offset terminates
          add.l   d1,(0,a3,d0.w)          ;add the base address to the
                                          ;absolute references to relocate
          bra.b   relocloop
branchtorelocated:
          jmp     (newaddress,a3)         ;execute from now on from the allocated
                                          ;memory space
newaddress:

;Allocate four 220K chunks of memory for the big buffer
          moveq   #16,d4
          rol.l   #4,d4     ;5632 = trackdisk buffer
          lea     (RamPtr+FastBoot,pc),a4
          movea.l a4,a3
;note that this is 220K +8 - the 8 provides protection of
;the blocks - extremely important!
          move.l  #225288,d5
          moveq   #3,d6
allocloop:
          move.l  d5,d0
          moveq   #0,d1     ;no special requirements
          SYS     AllocMem ;allocate 220K
          move.l  d0,(a3)
nomem:
          beq     notenoughboot     ;not enough memory, so load normally
          addq.l  #8,(a3)+ ;adjust so that the first 8 bytes are not
                            ;used by the driver
          dbra    d6,allocloop

;allocate 11 sector chip ram buffer for trackdisk II
          move.l  d4,d0
          moveq   #MEMF_CHIP,d1
          SYS     AllocMem ;allocate a 5,632 byte chip ram buffer
          movea.l d0,a5

          moveq   #0,d6     ;offset
          moveq   #3,d3     ;outer loop counter
          movea.l a4,a3
          move.l  (sp)+,a2 ;get back IO request pointer

;The disk read loop follows.

;d2 - IO request block pointer
;d3 - outer loop counter
;d4 - 5632 (constant)
;d5 - inner loop counter
;d6 - offset
;a1 - IO request block pointer (destroyed after each DoIO)
;a2 - active pointer to ram block
;a3 - active pointer to RAMPTR
;a4 - static pointer to RAMPTR
;a5 - pointer to chip ram

outerreadloop:
          movea.l (a3)+,a0
          moveq   #39,d5
innerreadloop:
          movea.l a2,a1
          move.l  a5,(IO_DATA,a1)
          move.l  d4,(IO_LENGTH,a1)
          move.l  d6,(IO_OFFSET,a1)
          move.w  #CMD_READ,(IO_COMMAND,a1)

          SYS     DoIO      ;read 11 sectors
          tst.l   d0        ;check error return
          bne     warmboot  ;reboot if an error occurred

          move.l  d4,d0
          movea.l a5,a0
```

Circle 121 on Reader Service card.

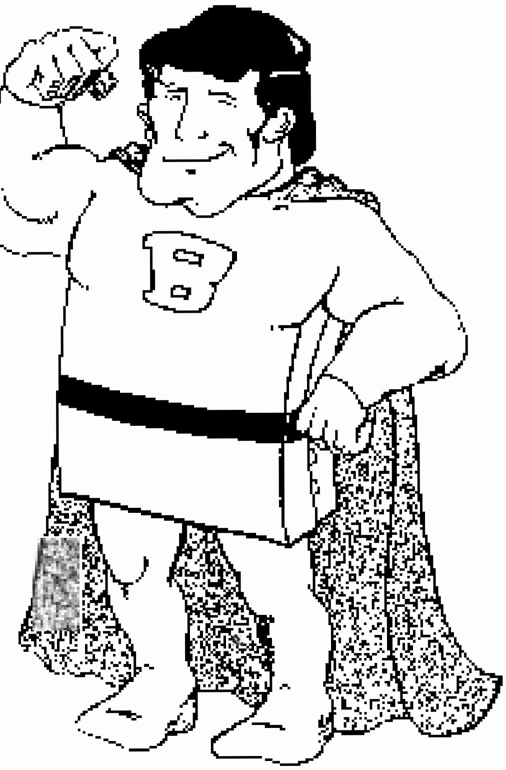
Circle 134 on Reader Service card.

```
copystraight:
        move.l  (a0)+,(a1)+
        subq.l  #1,d0
        bne.s   copystraight
        add.l   d4,d0
        dbra    d2,mainreadloop
        dbra    d3,mainreadloop

;Change the volume creation time of the RAD disk. If two
;devices have the same volume name AND creation time,
;AmigaDOS crashes.
        movea.l [illegible],a0
        addq.l  #1,([illegible],a0)     ;modify creation time (very slightly!)
        subq.l  #1,(20,a0)              ;fix checksum

;Deallocate chip memory buffer
        movea.l a5,a1
        move.l  d4,d0
        SYS     FreeMem

;Fill in KickMemPtr
        lea     ([illegible],pc),a0
        move.w  #1,(a0)+    ;number of entries in the table
        lea     (FastBoot,pc),a1
        move.l  a1,(a0)+    ;address of boot block & the kickmem list
        move.l  [illegible],(a0)+ ;length of above ([illegible] bytes)

        lea     ([illegible],pc),a1
        move.l  (KickMemPtr,a6),(a1)        ;chain
        move.l  a1,(KickMemPtr,a6)

;Fill in KickTagPtr
        lea     (MyKickTagPtr,pc),a0
        move.l  (KickTagPtr,a6),(4,a0)
        beq.s   .NoChain
        bset    #7,(4,a0)
.NoChain:
        move.l  a0,(KickTagPtr,a6)

;Fill in the proper kick checksum.
        SYS     SumKickData
        move.l  d0,(KickCheckSum,a6)

;Install the [illegible] in ColdCapture
        lea     ([illegible],pc),a0
        move.l  a0,(ColdCapture,a6)

;Recompute ExecBase checksum
        bsr.s   ExecBaseChecksum

;Fill in disk dimension structure
;This rather unusual approach saves several bytes...
        lea     ([illegible],a4),a0
        lea     (FillPatternInfo,pc),a1
        moveq   #7,d1
        moveq   #0,d0
fillpatchloop:
        move.b  (a1)+,d0
        move.b  (a1)+,(0,a0,d0.w)
        dbra    d1,fillpatchloop

;Add our driver to the exec device list and mount RAD
        lea     (Driver,pc),a1
        moveq   #-1,d0    ;"segment" (used here as a special flag)
        SYS     InitResident

        movem.l (sp)+,d2-d7/a2-a6
        bsr.b   [illegible]
        bsr.b   CheckVersion        ;version 1.3?
        bne.b   [illegible]         ;if not, then don't try to fix the bug
        moveq   #$30,d1
        add.l   d1,(a7)   ;skip over code to add an extra [illegible]
        rts

CheckVersion:
;Using this tiny subroutine saves 2 bytes overall (wow!)
        cmpi.w  #34,(LIB_VERSION,a6)        ;version 1.3?
        rts

ExecBaseChecksum:
;Enter with a pointer to ExecBase in a6
```

```
;uses d0,d1,a0

        lea     (BitfVec,a5),a0
        moveq   #13,d0
        moveq   #7,d7
strloop:
        addx.w  (a0)+,d1
        dbra    d0,strloop
        not.w   d1
        move.w  d1,(a0)
        rts

findres:
        movea.l (4).w,a6
        lea     (dos,pc),a1
        SYS     FindResident
        movea.l d0,a0
        movea.l ($16,a0),a0
Nope:
        moveq   #0,d0 ;this clearing of d0 is ABSOLUTELY needed for
                        ;"nope" to dummy driver function! !!
        rts

; //// AutoBoot structure ////
;Note that we take advantage of the holes in the bootcode
;structure by overlapping several structures. For a less
;confusing view of the bootcode, refer to the accompanying
;listing.

MyBootNode:
        dc.l    ' (C) 1992'       ;space for list linkage
        dc.b    16      ;type - this is crucial
        dc.b    127     ;priority - this gets changed to BootPriority
                        ;after a keyboard reset (see InitDriver)
relocX: dc.l    MyConfigDev       ;ConfigDev (required)
        dc.w            ;flags
        dc.l    'Dude'  ;ptr to devname, filled in later
MyConfigDev:
;7 16 bytes of nothing
MyKickTagPtr:
reloc2:
        dc.l    Driver
        dc.l    0       ;this zero is important, not just a space filler
[illegible]:
        lea     (4,a7),a7         ;pop
        bra.b   findres
BootPoint:
        dc.w    $FFFF
MyDiagArea:
        dc.b    16      ;important constant
Name:   dc.b    'DE',0
[illegible]:     dc.b    ;unused
;!!! NOTE: from here on is OBSCURED, as the bootable code
;MUST lie after the diagarea.
        dc.w    bootpoint-MyDiagArea
bootpoint:
;this is the standard bootblock code (almost)
        lea.l   Dosres
        jmp     (a0)

reloc3: dc.l    MyDiagArea

;////////////////////////////////////////////////////////////////

        even    ;just in case

ColdBoot:
        clr.l   (KickTagPtr,a6)   ;clear KickTagPtr
        jsr     SumKickData
        move.l  d0,(KickCheckSum,a6)        ;we do this to keep the 1MB Agnus fix
                                            ;intact even after we ColdReboot
                                            ;ColdBoot.
;We clear CoolCapture to ensure a restart of AKlue, if used.
        clr.l   (CoolCapture,a6)
        jsr     ExecBaseChecksum

noreboot:
        jsr     CheckVersion    ;compare Exec's LIB_VERSION with 37
        bls.b   NoReset
        SYS     ColdReboot      ;only available in 2.0+
MyVector:
          lea       (magicresetcode,pc),a5
          SYS       Supervisor
;never returns - jumps to magicresetcode

Name:     dc.b      'vdm.device',0
expansion:          dc.b      'expansion.library',0

          cnop      ;this MUST be longword aligned
;However, you should always try to arrange the code so that
;a dummy word is not needed - and it's not needed in this
;version. If you change the code size, keep this in mind.
magicresetcode:
          lea       (2),a0
          reset
          jmp (a0)
;end of reset code

agnusfix:
;fix bug in 1.2 and 1.3 Kickstarts - very important for
;>512K of CHIP. This code is equivalent to the "r" option of
;SetPatch.
          move.l    a0,(ColdCapture,a6)         ;ColdCapture is cleared upon entry - we
                                        ;must restore it
          movea.l   (2,a7),a5
          cmpi.l    #$07FC0096,(816,a5)         ;crude version check
          bne.s     skipagnusfix        ;not 1.2 or 1.3, so don't do anything at all
          jmp       (2,a5) ;the 2 offset skips over the buggy code
skipagnusfix:
          jmp       (a5)

relocinfo:
;table used to relocate the absolute addresses used in the
;driver

          dc.w      reloc1,reloc2,reloc3,reloc4,reloc5,reloc6,reloc7
          dc.w      reloc8,reloc9,reloc10,reloc11,reloc12,reloc13

Table2:
          dc.w      0         ;end of relocation info (and start of Table2)
          dc.w      10
reloc4:   dc.l      Functions
reloc5:   dc.l      DataTable
reloc6:   dc.l      InitDriver          ;tell the exec about our device

;*** ram disk driver ***
Driver:
Table1:
          dc.w      $4AFC
reloc7:   dc.l      Table1
reloc8:   dc.l      EndCode
          dc.b      RTF_AUTOINIT+RTF_COLDSTART
          dc.b      0         ;version
          dc.b      NT_DEVICE
          dc.b      20        ;priority
reloc9:   dc.l      Name
          dc.l      0         ;who needs an IDString, anyway?
reloc10:  dc.l      Table2

Functions:
          dc.w -1   ;this indicates that the following are offsets,
                    ;rather than absolute addresses
          dc.w      Noop-Functions
          dc.w      Noop-Functions
          dc.w      Nope-Functions
          dc.w      Nope-Functions
          dc.w      BeginIO-Functions
          dc.w      Nope-Functions
          dc.w      -         ;this delineates the end of code table

DataTable:
;initialization data for the device structure

  INITBYTE LN_TYPE,NT_DEVICE

; INITLONG LN_NAME,Name
          dc.w      $C00A
reloc11:  dc.l      Name

          dc.l      0         ;end of init-struct table
          dc.w      0         ;space filler
```

```
AddMem:
;Special code to support the two most popular A1000 hacks
;This code is perfectly harmless on other machines.

;Note that this routine requires the ExecBase ptr to be in
;a6.

;Toggle the motor (which also toggles the LED on drives
;                  supplied by C= and most 3rd party drives)
;This enables the extra 512K at $80,000 for some A1000
;owners.

          moveq     #3,d0
          moveq     #7,d1
          move.l    #$bfd100,a0
          bclr      d1,(a0)   ;motor on
          bset      d0,(a0)
          bclr      d0,(a0)

          bset      d1,(a0)   ;motor off
          bset      d0,(a0)
          bclr      d0,(a0)

addmem1:
          [illegible]  [illegible]
;Check for the presence of memory at $80,000
;Note that the test is nondestructive
          move.l    #$80000,a0
          move.w    (a0),d0
          move.w    #$2931,d3          ;test word
          move.w    d3,(a0)
          cmp.w     (a0),d3
          bne.s     a1000kickads
          move.w    d0,(a0)

;Add our extra memory
          moveq     #4,d0
          swap      d0        ;load $40,000 (262,144)
          bra.s     addmemsub

a1000kickads:
          moveq     #8,d0
          swap      d0        ;load $80,000 (524,288)
          move.l    d0,a0

;Sanity check: If MaxLocMem is not $80,000, don't AddMem at
;$80,000. This case occurs when using a 8372 Agnus.
          cmp.l     (MaxLocMem,a6),d0
          bne.s     skipaddmem

;Check for memory at $80,000
          move.w    (a0),d1   ;save
          move.w    d3,(a0)
          cmp.w     (a0),d3
          beq.s     skipaddmem
          move.w    d1,(a0)   ;restore

;d0.l (length) and a0.l (starting address) must be set first
addmemsub:
          moveq     #MEMF_PUBLIC!MEMF_FAST,d1  ;attributes
          moveq     #0,d2     ;priority
          suba.l    a1,a1     ;name
          SYS       AddMemList
skipaddmem:
          clr.w     (a0)
skipaddmem1:
          rts

InitDriver:
          movem.l   d0-d7/a0-a6,-(sp)
          move.l    (4).w,a6

;the "segment" is in d0
;a special trick (like DOS)
          move.l    d0,d0 ;set the flags (no other purpose)
          bne.s     wasMount

          bsr       CheckVersion
          bne.s     init1
          lea       (ResetCount,pc),a2
```

```
         addq.w   #1,(a0)
         beq.s    .02

.01:     lea      19+MyBootNode,pc),a0
         move.b   #BootPriority,(a0)          ;set boot priority. [illegible] is priority
                                              three.
.02:
         btst     #7,$BFE001        ;check port 2 fire button
         beq      CoolReset ;go if pressed

         bsr      AddMem
;Call AllocAbs to reclaim our memory
         lea      [illegible],pc),a4
         moveq    #3,d5

;The next instruction was broken apart logically so that
;the operand could be used elsewhere.
;Equivalent to: move.l #225280,d7
         dc.w     $2E3C
ChunkSize         dc.l      225280    ;220K

AbsAllocLoop:
         movea.l  (a2)+,a1
         move.l   d7,d0
         SYS      AllocAbs
         dbra     d6,AbsAllocLoop

JustMount:
         movea.l  (4).w,a6
         lea      (ExpansionName,pc),a1
         SYS      OldOpenLibrary      ;this clears D0!
         movea.l  d0,a6
         lea      (parameterPkt,pc),a0
         SYS      MakeDosNode
         lea      (eb_MountList,a6),a0        ;pointer to eb_MountList
         movea.l  (4).w,a6
         lea      (MyBootNode,pc),a1          ;pointer to a BootNode structure
         move.l   d0,(16,a1)          ;place DosNode pointer in BootNode structure
         SYS      Enqueue   ;add BootNode to eb_MountList
enddriverinit:
         movem.l  (sp)+,d0-d7/a0-a6
         rts

BeginIO:
         movem.l  d0-d7/a0-a6,-(sp)
         movea.l  (4).w,a6
         clr.b    (IO_ERROR,a1)       ;no error - in fact we never report an error
         move.w   (IO_COMMAND,a1),d7
         movem.l  (IO_LENGTH,a1),d4-d6         ;read IO_LENGTH, IO_DATA, & IO_OFFSET
         movea.l  d4,d1
         moveq    #9,d6
         [illegible] #9,d6
         moveq    #1,d4
;d4   IO_LENGTH; d4 - IO_DATA; d5 = IO_OFFSET; d6 = 512
;d7 = IO_COMMAND; d4 = 2
         move.l   d4,(IO_ACTUAL,a1) ;IO_ACTUAL = IO_LENGTH
         [illegible] #11,d7
         cmp.w    #4,d7
         beq.s    ReadWrite
         cmp.w    #3,d7
         beq.s    ReadWrite
         cmp.w    #11,d7
         beq.s    ReadWrite           ;format

         clr.l    (IO_ACTUAL,a1)      ;IO_ACTUAL - this is definitely needed!
                                      ;otherwise we get a "no disk in drive" message!

;If command is unknown, ignore it
TermReturn:
         btst     #IOB_QUICK,(IO_FLAGS,a1)
         bne.s    TermEnd
         SYS      ReplyMsg
TermEnd:
         bra.s    enddriverinit

ReadWrite:

;The following chunk of code computes the address in RAM
;corresponding to IO_OFFSET (in d6).
```



```
          move.l    d5,d0
          moveq     #-4,d1
.compute
          addq.l    #4,d1
          movea.l   d0,a0
          sub.l     (ChunkSize,pc),d2
          [illegible]     ;compute sum, unsigned

          lea       (pcml+FastBoot,pc),a2
          adda.l    (a2,d1.w),a0

          move.l    d5,d0
increaseWriteLoop:
          cmp.w     d4,d7     ;check end
          blt.s     writeit

          move.l    (a0)+,(a6)-
          subq.l    #4,d0
          bne.s     increaseWriteLoop
ReadWrite1:
          add.l     d6,d5     ;add 512 to the current offset
          sub.l     d6,d3     ;and take 512 from the length
          bhi.s     ReadWrite
          bra.s     endReturn

writeit:
          move.l    (a4)-,(a6)-
          subq.l    #4,d0
          bne.s     writeit
          bra.s     ReadWrite1

FillPacketInfo: ;note 8 pairs
;The first byte in each pair is a byte offset into the
;parampacket. The second byte is the value to set
;there.
          dc.b      11,11
          dc.b      15,128
          dc.b      23,2
          dc.b      27,1
          dc.b      31,11
          dc.b      35,2
          dc.b      51,79
          dc.b      55,5

parampacket:
reloc12:
dosnameptr:         dc.l      Name
reloc13:
execnameptr:        dc.l      Name

;The following is the structure that will be generated dynamically using
;FillPacketInfo.

;         dc.l      0         ;unit number (not very important)
;         dc.l      0         ;OpenDevice flags (not used in this case)
;         dc.l      11        ;upper bound of this table, in longwords
;         dc.l      128       ;number of longwords in a block (512/4)
;         dc.l      0         ;sector origin (never used)
;         dc.l      2         ;number of surfaces
;         dc.l      1         ;sectors per logical block (cluster size) - never used
;         dc.l      11        ;sectors per track
;         dc.l      2         ;reserved blocks - 2 boot blocks
;         dc.l      0         ;never used
;         dc.l      0         ;interleave
;         dc.l      0         ;lower cylinder
;         dc.l      79        ;upper cylinder
;         dc.l      5         ;number of buffers

EndCode:

;Note that there are 24 bytes after the disk dimension structure
;used for holding the KickMaster information starting at
;offset 1072 and ending at offset 1095 non-inclusive.

;The four handle pointers start at offset 1096 and end at
;1112. These pointers are the result of the four AllocMem
;calls, each requesting 220K.

          END
```

☑

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! ◆

Subscribers can purchase freely redistributable disks at bulk rate discount prices!**

This unbeatable offer includes all Fred Fish, AMICUS, and AC disks

(see the back of this issue for recent Fred Fish additions, and 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 insert in the issue.

* 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 warranties 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 programmer archivist to continue his outstanding work.

AmigaDOS for Programmers

Exploring DOS Library Calls and Features

by Bruno Costa

The Amiga system software is clearly layered into multiple levels of complexity with well-defined purposes. It is completely implemented as calls to memory-resident or disk-resident system libraries that contain virtually all the functions and data a program needs to control every feature of the machine. Managing the lowest level characteristics is Exec, the multitasking kernel where things like task switching, messages, signals, processor exceptions and device-task communications are handled. There is also the graphics library that controls custom chips and basic graphics. The layers library provides a way to share graphic screens between multiple tasks, "slicing" the display into layers (very similar to windows). At a higher level, Intuition uses all of these to implement the menus/windows/gadgets graphical user interface. "But what about AmigaDOS?", you may ask. Well. .

Exec provides some tools that allow multiple programs to communicate with devices by sending messages back and forth, but the Exec devices just read and write blocks of data, we need a way to logically divide the space of each disk device into directories and files. That's one of the tasks of AmigaDOS, in addition to file formats, protection bits, processes (an enlarged task), device handlers, virtual device name assignment and a powerful interface to all of these activities. Normally, when you use the CLI, you do not call AmigaDOS directly (this is only possible through programs). Rather, you load files in executable format (executable programs) which, in turn, call AmigaDOS.

This article is not intended to be a full reference or an entry-level tutorial. Rather, it represents a collection of personal findings and tips on how to call AmigaDOS from inside your programs. Combined with the *AmigaDOS Developer's Manual*, the *AmigaDOS Technical Reference Manual*, and the example programs provided at the end of the article, I hope to make clear how—and sometimes why—things work (or do not). If you want to delete files, find out file sizes, attributes or the amount of disk space free, create or read directories and even run processes from inside your programs, read on.

Files

AmigaDOS provides a basic set of file-handling functions that range from simple read/write operations to low-level information and locking protocols that allow simultaneous file access by concurrent processes. There are two ways under AmigaDOS to identify files: locks and handles.

Locks are used to notify the system that you want to access a file (or directory); they thus forbid another process to modify that file while you are reading it, or to read it while you are writing. A lock also provides a unique identification for a file that is frequently used by library functions to describe which file you are referencing. A lock can be obtained explicitly by a call to Lock (filename, access), where filename is a regular file name and access is either ACCESS_READ (same as SHARED_LOCK) or ACCESS_WRITE (same as EXCLUSIVE_LOCK). Lock( ) will fail, returning a NULL value, if the given file is not accessible (e.g., the file does not exist or is already exclusively locked), so it can be used to check it. When you are done with the file access you must call Unlock (lock), where lock is a valid lock returned by the previous Lock( ) call.

In case you want to read or write to a file you will need something more robust and meaningful than a lock: a handle. Handles are obtained explicitly by a call to Open (filename, access), where access is now either MODE_NEWFILE or MODE_OLDFILE, to create a new file or use an existing one. Once you have a handle, you will be able to perform read and write operations on a file: this can be achieved by Read (handle, buffer, length) or Write (handle, buffer, length), where buffer is a piece of memory of length bytes where you want the data to be written to or read from. Both functions return the actual number of bytes processed, which can be less than expected (due to end-of-file or disk-full conditions, for instance), or -1 in case of an error. When you are done with the file you must call Close (handle). Note that you needn't lock a file for reading or writing when you use Open( ), as this is done automatically.

A major source of AmigaDOS power comes from the transparency with which the system treats devices as different as a hard disk file, an Intuition window or the serial port (there is now even the SPEAK: device!). Inside your program you may call Open ("SER:", MODE_NEWFILE) for instance and, if you use the resulting file handle, you will be able to read from or write to the serial port exactly as though you were using a normal file. Normally these special files cannot be read more than once, since data is not stored inside them permanently. They may be seen as a continuous, ever-flowing stream of bytes (either incoming or outgoing) that, once read, is completely discarded (unless you store it somewhere else).

On the other hand, there are some files (the so-called random access files) where any particular portion can be read and reread many times (disk files are always of this type). For special applications you may need to move back and forth through a random access file, especially to go to a particular place in the file at once. This can be done using the Seek (handle, position, reference) call. Position is an integer (positive or negative) specifying to which particular byte position you want to move; reference tells the system from where you are counting the given position. It may be one of OFFSET_BEGINNING (from the first position in file), OFFSET_END (from the last byte in file) or OFFSET_CURRENT (from the last position you read or wrote to). Seek( ) returns the previous file position counted from the start of file or -1 in case of error (e.g., the file is not random accessible or the specified position is outside the existing file). Note that Read( ) and Write( ) do their work at the current file position (modifiable at any time by the Seek( ) call), making it possible to read (or write) arbitrary portions of a file as many times as you want. Here are some examples:

```
Seek (file, 0, OFFSET_BEGINNING)       ; "rewinds" a file
pos = Seek (file, 0, OFFSET_CURRENT)   ; returns current
                                       ; position leaving
                                       ; file unchanged
Seek (file, 0, OFFSET_END)             ; goes to end of
                                       ; file (useful for
                                       ; appending data)
```

The ability to move directly to particular positions in a file may be very useful for indexing files. You could use it to write a help facility for a program. In addition to the help file (a plain text file), you would have an index file containing the positions of each help page in the help file. If the user needed the third help page, for instance, your program would read the third number in the index file; this number would determine the place in the help file where the third page started. Then you would call Seek( ) for the help file with the index number you just read, and the file pointer would move to the exact position where the page started. You should then call Read( ) to load the help page and display it afterwards. Note that this approach will make access to the file much faster than the conventional sequential access (with the latter method you would need to read the whole file to find each particular page).

There are two other calls very useful for file manipulation: DeleteFile (filename) and Rename (oldname, newname). Their names are more than self-explanatory. Both functions return TRUE when successfully executed and FALSE when there is an error (if the disk is write-protected, for instance).

Now you should know enough to check some of the examples. The file copy program, named cp.c, is a good, simple example (but not so powerful, since it does just a subset of what the AmigaDOS copy command does). It simply reads from the first file and writes to the second as long as it can. There is also another example, named del.c, that shows how to call DeleteFile( ) properly. Experiment with the programs—but please, not on your hard disk or on the only copy of that nice utility (the programs work, but who knows?).

Standard I/O

I mentioned that processes are handled by DOS, but did not explain how or why. A process is a task that knows what files, paths, and current directory are; a normal task just knows how to handle processor and multitasking related events. If you want to call AmigaDOS you must do so from inside a process, because knowledge of these things is needed by most of the routines. If you run your program normally (from the CLI or Workbench) that will not be a problem since it will be spawned as a true process. A process also knows a standard place to read data and another to write. This is very similar to the C language's stdin and stdout, which are implemented on top of these process features.

To obtain the standard file handles you may call Input( ) or Output( ) which take no parameters and return the respective handles. You may use these handles at will (as long as you write to output and read from input, of course) and you don't have to close them (in fact, you can't). Standard I/O provides a simpler way to access files (since you don't have to open or close them), and for certain programs they are a natural definition. These special programs are called filters because they read a stream of bytes, do some processing (or filtering) and write them to output. When there is an easy way to combine multiple filters, sending the output from one as input to the next, they become a powerful tool. One way to do this concatenation is called piping, a concept present in Unix and MS-DOS machines but still lacking in the Commodore Shell. To use pipes, you would simply type two (or more) program names separated by a piping symbol, indicating that both programs should be run simultaneously, piping the output of the first as input to the next. Note that, to be usable in pipelines, programs must be able to read from Input( ) and write to Output( ).

Now have a look at the second example program, sidecp.c, a variation on the previous one that copies everything from its standard input to its standard output, until end-of file is reached. When used with the CLI or Shell I/O redirection (the > and < shell operators) it may be used with the same effect as cp.c:

```
        cp file1 file2                 ; copies file1 to file2
        sidecp <file1 >file2           ; also copies file1 to file2
        sidecp                         ; copies from keyboard to
screen until EOF (Ctrl-\) is typed
```

You can see by the last example that if no redirection is used the default input is the keyboard and the default output is the console window. If your program is run through Workbench, these standard handles are normally invalid (Input( ) and Output( ) return NULL), but if you wish to print messages you may open a console window yourself (note that your compiler may open a console window for you automatically).

Attributes

AmigaDOS stores files as a linked list of disk blocks, each containing some data and the number of the next block. The first of these blocks is the header of the file, and it contains several attributes of the file: protection bits, name, size, an eighty-character comment and the last modification date. A directory is also stored as one of these header blocks, with the size equal to zero and no data blocks. All this information can be obtained transparently using the Examine (lock, fileinfo) library call, where lock is a lock on the target object (either a file or directory) and fileinfo is a pointer to a previously allocated FileInfoBlock structure. If Examine( ) returns TRUE, the structure was filled with the correct information on that object; otherwise an error occurred. The FileInfoBlock structure is defined in libraries/dos.h as:

```
struct FileInfoBlock {
  LONG        fib_DiskKey;         /* disk block number ID */
  LONG        fib_DirEntryType;    /* specify file or dir */
  char        fib_FileName[108];   /* name of the file */
  LONG        fib_Protection;      /* protection mask */
  LONG        fib_EntryType;
  LONG        fib_Size;            /* size of file (bytes) */
  LONG        fib_NumBlocks;       /* size of file (blocks) */
  struct DateStamp fib_Date;       /* modification date */
  char        fib_Comment[80];     /* file comment */
  char        fib_Reserved[36];    /* future extensions */
};
```

The first field, fib_DiskKey, is normally the number of the disk block where this object (file or directory) starts, but it may be any unique integer identifier of the object on a particular disk (the RAM disk, for instance, does not have real disk blocks). If the fib_DirEntryType field is positive the object in question is a directory; otherwise it is a simple file. Both files and directories have the fib_FileName and fib_Comment fields filled with the correct zero terminated strings with up to 30 and 80 characters respectively. The sizes, used only for files, are simply integers containing the number of bytes and number of blocks occupied on disk. The protection mask in fib_Protection will be explained later with the SetProtection( ) call. The field fib_Date contains the date the file was last modified in the following format:

```
struct DateStamp {
  LONG ds_Days;          /* Number of days since 1/1/1978 */
  LONG ds_Minute;        /* Number of minutes past midnight */
  LONG ds_Tick;          /* Number of 1/50 secs past minute */
};
```

Now have a look at the example program examine.c. It is simply an Examine( ) CLI interface, allowing you to examine any file or directory by giving its name as an argument to the program. The output is a C-like formatted FileInfoBlock structure with the fields and corresponding values shown. Use it to understand exactly what the fields mean and how different devices may respond to the Examine( ) request.

Some of the file attributes can be modified using corresponding library functions. The SetProtection (filename, mask) call returns TRUE if the new protection bits of the file filename are set to mask. Currently the protection bits may be a combination of script, pure, archive, read, write, execute and delete permissions, and they have the following order and meaning.

S = 1 if the file is a shell script, else 0.
P = 1 if the file is pure (can be made resident), else 0.
A = 1 if the file was archived (for hard disk backups),else 0.
R = 1 if the file cannot be read, else 0.
W = 1 if the file cannot be written to, else 0.
E = 1 if the file cannot be executed, else 0.
D = 1 if the file cannot be deleted, else 0.

If, for instance, a file has protection bits ----RWED (0000000) the mask is 0x00; for S—R-E- (1000101) mask is 0x45; for SPARWED (1110000) mask is 0x70.

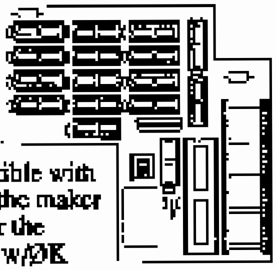
The comment of a file can be changed using the SetComment (filename, comment) call that returns TRUE if the comment of filename was successfully changed.

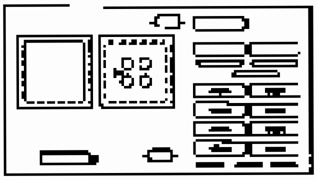
Note that the Rename( ) call modifies the fib_FileName field. Other file attributes can also be changed, but not directly. The date and sizes of the file are modified if you write to it. You may read one byte and write it back to the same place to modify a file date but not its contents, as the program touch.c does. Touch.c is used to set the date of a file to the current date, and is also an example of the Seek( ) call since it is used to move back to the start of the file to write the byte. Note that there is a better way to change a file's date (the method used by the AmigaDOS SetDate command) but, unfortunately, there is no direct library call to do it.

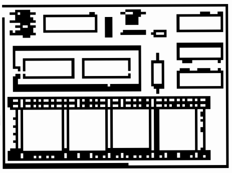
You saw that Examine( ) allows you to gather information on files, but to get information on a disk device status you need to call Info (lock, infodata), where lock is a lock on any file or directory in the device and infodata is a pointer to an allocated InfoData structure. Info( ) returns TRUE if the information was successfully obtained and the InfoData structure properly filled. This structure is defined in libraries/dos.h as:

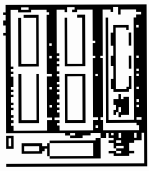
```
struct InfoData {
  LONG      id_NumSoftErrors;    /* Number of soft errors on disk */
  LONG      id_UnitNumber;       /* Which unit disk is */
                                 /* (was) mounted on */
  LONG      id_DiskState;        /* Validation (see below) */
  LONG      id_NumBlocks;        /* Number of blocks on disk */
  LONG      id_NumBlocksUsed;    /* Number of block in use */
  LONG      id_BytesPerBlock;    /* Number of bytes on each block */
  LONG      id_DiskType;         /* Disk type code (see below) */
  BPTR      id_VolumeNode;
  LONG      id_InUse;            /* Flag, zero if not in use */
};
```

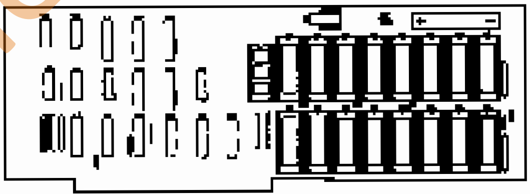
Most of the information in the InfoData structure is presented to you by the info CLI command. The number of software errors, for instance, is exactly the same as presented by info. The id_DiskType is the disk ID present in the first four bytes of the disk and may be one of the following: ID_NO_DISK_PRESENT, ID_UNREADABLE_DISK, ID_DOS_DISK, ID_NOT_REALLY_DOS or ID_KICKSTART_DISK. The id_DiskState indicates the disk validation status (disk validation is a testing process executed every time you insert a disk in a drive): ID_WRITE_PROTECTED if the disk could not be validated; ID_VALIDATING if the disk is currently being validated; or ID_VALIDATED if the disk was correctly validated and is thus write enabled. The most useful information you can obtain from the Info call regards disk size and space, which can be calculated as follows:











```
TotalDiskSize = id_NumBlocks * id_BytesPerBlock
UsedDiskSpace   id_NumBlocksUsed * id_BytesPerBlock
FreeDiskSpace   TotalDiskSize - UsedDiskSpace
```

The example program space.c, when given the name of a disk device or any file in it, shows the used, available and total disk space in that device by calling Info( ). You can use it as an example of how to display the disk available space in a file requester, for instance.

Directories

When you use the CLI, the concept of a "current directory" is always present: it is a special directory where you "are", and no path names are required to refer to files inside it. When a process is created it has a current directory, normally inherited from the process that spawned it. If it came from the CLI, the current directory is exactly the same as the CLI's when the process was begun (if you use the cd command after you start a process, the current directory of this process will not be modified). If it came from Workbench, the initial current directory is the drawer where the program was stored. If you wish to change the current directory of your program, you may call CurrentDir (lock), where lock is a lock on the new current directory, and the old directory is returned from the function. Note that the current directory of your process will be modified (but nothing else), because the current directory is a property of each process. If you subsequently spawn a new process it will inherit this new current directory, but previously spawned processes will not be affected.

Directories under AmigaDOS (and many other operating systems) are hierarchically organized as an upside-down tree, in such a way that a particular directory may have as many subdirectories as the user wants, but there is only one way up. The subdirectories may be called "child" directories, and they have a corresponding unique "parent". To traverse a directory tree, one must be able to know the available ways down (by reading the directory) and how to come back up. The latter task is accomplished by the ParentDir (lock) call, which returns a lock on the parent directory of the file or directory given by lock. Although not very useful, there are some cases when this call cannot be substituted in any way, as in the example program full.c. The program is basically a main( ) provided to test a rather useful routine that returns the path name of a file (or directory) given by a lock. It continues to call ParentDir( ) to get a lock on the parent directory and Examine( ) to know its name, until the root of the filing system is reached. The last name obtained before ParentDir( ) returns NULL is the disk volume name. The directories are concatenated with intermixed slashes and separated from the disk name by a colon (:), building a complete path name. Note that a similar function is provided in the standard Lattice C library (no source, of course) with the name getpath( ).

If you wish to create a directory on disk, you should call CreateDir (dirname). The string dirname may contain complete or partial AmigaDOS path names. If possible, the routine will create the specified directory and return a shared lock on it. Normally you will unlock it immediately, unless you need to be sure that the directory will still be there for a certain operation to take place. Note that, in any case, you must unlock it sooner or later. If an error occurs, this function will return a NULL value.

Now that you know how to create and move inside a directory hierarchy, you just need to learn how to read the contents of a directory. The ExNext (lock, info) call is provided specifically for this purpose. It returns TRUE if it succeeds, and takes two parameters: Lock is a lock on the directory you are reading, and info is a pointer to a FileInfoBlock structure (described in the first part of the article and defined in libraries/dos.h). The FileInfoBlock must be previously initialized with data from the directory you are reading by a call to Examine (lock, info). After that you should look inside info to see if the lock is indeed related to a directory, and not to a file. Each time you call ExNext( ) it returns information on a particular object in the directory: when you call it again, it uses data inside info to determine which object is to be examined next. Note that the information provided is quite comprehensive and may be used in any way, but you should not modify fields inside info because ExNext( ) may need them the next time you call it. You continue to examine each object in the directory until no more entries are left and ExNext( ) returns a FALSE condition.

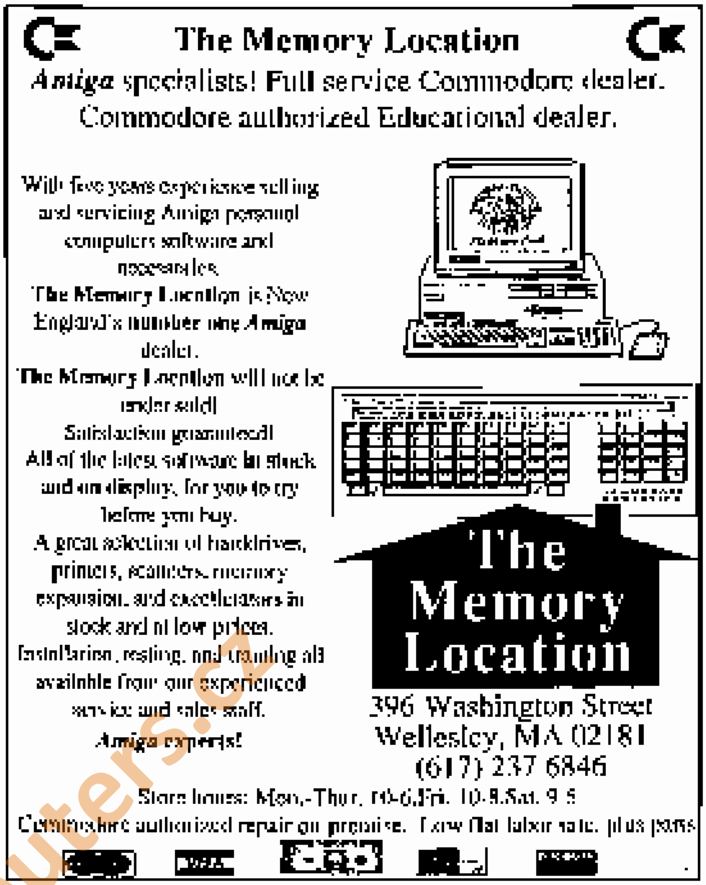
The complete procedure needed to read a directory is detailed in the example program ls.c. It lists the complete information on a given directory, and uses Examine( ) and ExNext( ) in the manner described above. The information is presented in a somewhat rough format, and things like the protection bits and the date are not converted in humanly legible form. The program is very simple, but can be improved to read arbitrary directories in a general form. This might be done by writing a routine that returns a list of file information blocks which could be used by many different tasks where it is necessary to read a directory, including wild card expansion and file requesters.

Process Handling

Although very different to the end user, there are some similarities between the CLI and Workbench. Both provide a user interface to some of the features of the underlying operating system: tools to manipulate files and run programs. In a strict sense both are "shells" over the operating system: the CLI is a classic command line shell , with powerful concepts inherited from other operating systems (like Unix); Workbench is a graphical shell, easier and more intuitive to use. In fact, there are some commercial programs (like CLIMate, DiskMaster and DirUtil) that can be also considered shells. It is possible to create your own shell, with any interface you desire (e.g., graphical, touch-sensitive, voice-driven) using the concepts provided here and in the reference material (although it may be quite complicated to write a truly complete and useful one). An example will be provided at the end of this section.

AmigaDOS lacks easy-to-use, complete process spawning and handling functions. The two calls available to create new processes are very poorly documented, limited and full of bugs in such a way that only after considerable experimentation can one make any use of them. It is likely that developers write their own substitutes for these functions when truly correct and complete process handling is needed. To create some simple processes, consider the Execute (cmdline, input, output) call. It attempts to execute the CLI command (given with any needed parameters in cmdline), and returns a TRUE condition if it succeeds. The string cmdline may also contain any standard I/O redirection specifications. In fact, just about any input that can be typed in the standard CLI can be executed by this call (at least theoretically). Note that Execute( ) won't search your commands throughout the current path—you will have to provide full path names to be sure it will find them.

Although output is exactly what you would expect (a handle to specify where the command's output should be written to), input is a handle specifying where to get more commands (similar to a batch file); usually it will be NULL. Note that there seems to be no way to provide simple standard input to programs run through Execute( ). If output is NULL, the current output (the handle returned by Output( )) will be used. However this is not all Execute( ) can do—it has one more exotic use. If you give it a NULL cmdline, a NULL output and a console window handle as input (like the result of Open ("CON:0/0/640/200/My CLI", MODE_NEWFILE)), a new interactive CLI will be created! It will only terminate when the



user types the EndCLI command inside it; only then will control be returned to your program (you should then close the window you opened). Believe it or not, this call will not work if the Run command is not present in the commands directory (C:). Note that there is no way to determine the return value of the commands run by Execute( ), and that your program will be frozen until the process you spawned terminates.

Just for the sake of completeness, a simple and limited command-line shell (well, kind of) is provided as sh.c. It will read lines from a window and pass them directly to Execute( ) that will (hopefully) do what you want, writing the output normally to the window. Note that path, cd and many other commands that modify fields inside the CLI control structure will not work, because Execute( ) runs commands in a sub-CLI. It allocates new CLI and process control structures, fills them properly and spawns the program as a stand-alone process. If a program modifies any field in the CLI control structure, only the sub-CLI fields will be effectively changed because the program being run simply does not know who ran it. When the program terminates, the modified sub-CLI will be discarded along with all changes made to it.

Error Handling

Error handling plays a major role in the development of professional software. Good programs should always be entirely operative and trustworthy, even in the worst situations. AmigaDOS provides usual error-handling features, and almost all library calls are able to notify error conditions. Usually, these errors are only signaled as flags (an error occurred but no real information on the nature of it is provided). The IoErr( ) call supplies this information, simply returning the current error state represented by an integer. Note that the error codes supplied by IoErr( ) always refer to the last library call and that it is meant to be called only after an error has happened. The error codes are documented in the include file libraries/dos.h, and symbolical names are given to them. If, for instance, you are reading a directory and there are no more files inside it (the end-of-directory condition mentioned above), ExNext( ) returns an error flag and, if you call IoErr( ) just after this, it will return ERROR_NO_MORE_ENTRIES. If you examine the ls.c source, you will see how easy this simple case error handling is, but, as programs grow and more complex situations appear, it may become more and more difficult.

You may have noted the emphasis placed on freeing what you allocate, closing what you open and returning what you get. This is done because AmigaDOS does not track resources—that is, it simply does not know which files you have opened or locked. If you exit from your program without freeing these resources, they will be lost forever: locked or opened files will be undeletable, unwritable and sometimes unreadable until you reboot your machine. This is a serious programming error, and you should be very careful when tracking the resources you get from the system (not only from AmigaDOS, but also allocated memory, opened libraries and so on), making sure they will be returned or closed by your program in virtually every situation that may arise.

Libraries

Although libraries are handled by Exec, a thorough understanding of how they work may help a good deal in several programming tasks. There are two basic kinds of libraries: linked and shared.

Linked libraries are those that come with your compiler and you use with the linker, the normal kind of library, similar to those on almost every other computer. They are basically a collection of object modules produced by a compiler, where the linker knows how to find particular routines and variables. The exact effect of a library could be achieved by giving dozens of object files to the linker. The important thing to know is that, when you link a routine in your code (when you call printf( ), for instance), the program size increases proportionally. Actually, a copy of that routine is placed inside the compiled code of every program that calls it.

Shared libraries are those inside the LIBS: directory (like translator.library) or resident in read-only memory (like dos.library). We call them shared because there is a single copy of each library routine even if multiple programs are calling it at the same time (they share the common code amongst them). If you call a shared library in a program it won't add much to the code size, and the size of such a program may be dramatically less compared to those linked with normal libraries.

Shared libraries are unique to the Amiga and are made up by some data space and a series of library vectors, or assembly language jump instructions, which point to pieces of code of that library that perform determined functions. You (and the linker) simply don't know where in memory a particular library will be since, each time you turn on your computer, they may be loaded in a different place. Only Exec can determine where the libraries are located, and to obtain the base address of a library you should call the OpenLibrary( ) Exec function. These library base pointers must be stored in some fixed name variables, where the compiler looks to find them (IntuitionBase and DOSBase are some examples). It may seem strange but, if you use a base pointer variable name that is different from the standard one, the compiler will not know where the library is and the program will not work (admittedly this was quite hard to find out). The functions in a library are "numbered" and, although you don't need to know these numbers, your compiler does, and they are very well documented (check the *Amiga ROM Kernel Reference Manual: Libraries and Devices*). As a hypothetical example, if you call the function Bar( ), which your compiler knows is the function number 5 of the foo.library, generated code is simply something like: "jump to the function pointed to by the fifth vector counted from FooBase."

AmigaDOS is implemented as a shared library (dos.library) that is memory-resident, but you needn't worry. The compiler-supplied startup code (that little thing that runs and sets things up before main( )) normally opens the dos.library automatically and places the base address in DOSBase. If you wish, you can open the library yourself, as it will not harm the system (as long as you close the library with CloseLibrary (DOSBase) afterwards). Although you may need some simple assembly language programming, it is possible to write your own shared libraries and even modify existing ones, but that's another story.

Conclusion

Hopefully the concepts presented here will help you to write your own AmigaDOS applications and utilities. You might have already found some limitations when experimenting with the DOS library, but you will surely find even more if you try to use it for a more complex task. Particularly, I think that the AmigaDOS programmer's dream is already real, in the form of the ARP utilities and library. ARP stands for AmigaDOS Resource Project. It is a group effort headed by Charlie Heath (of MicroSmiths, Inc.) and all their work can be freely redistributed. ARP provides easier access to the AmigaDOS device list, easier directory reading, wildcard patterns, date conversion routines, argument parsing facilities, a ready-to-use and flexible file requester, perfect synchronous and asynchronous process creating calls, a complete shell including piping and much more. You may obtain a copy of the latest ARP release (version 1.3) by sending $5.00 (postage and handling expenses) to the following address:

ARP Support
c/o MicroSmiths, Inc.
P.O. Box 561
Cambridge, MA 02140

Be sure to obtain the complete release, with the programmer's documentation, examples and proper compiler utilities (startup code, include files and libraries). If you want to do serious programming work under AmigaDOS it will be a valuable aid indeed. ✓

About The Author

Bruno Costa is a computer engineering student and works with computer graphics at the IMPA (Institute for Pure and Applied Math) in Rio de Janeiro, Brazil. He has owned Amiga computers since 1987.

A Unique Input Device

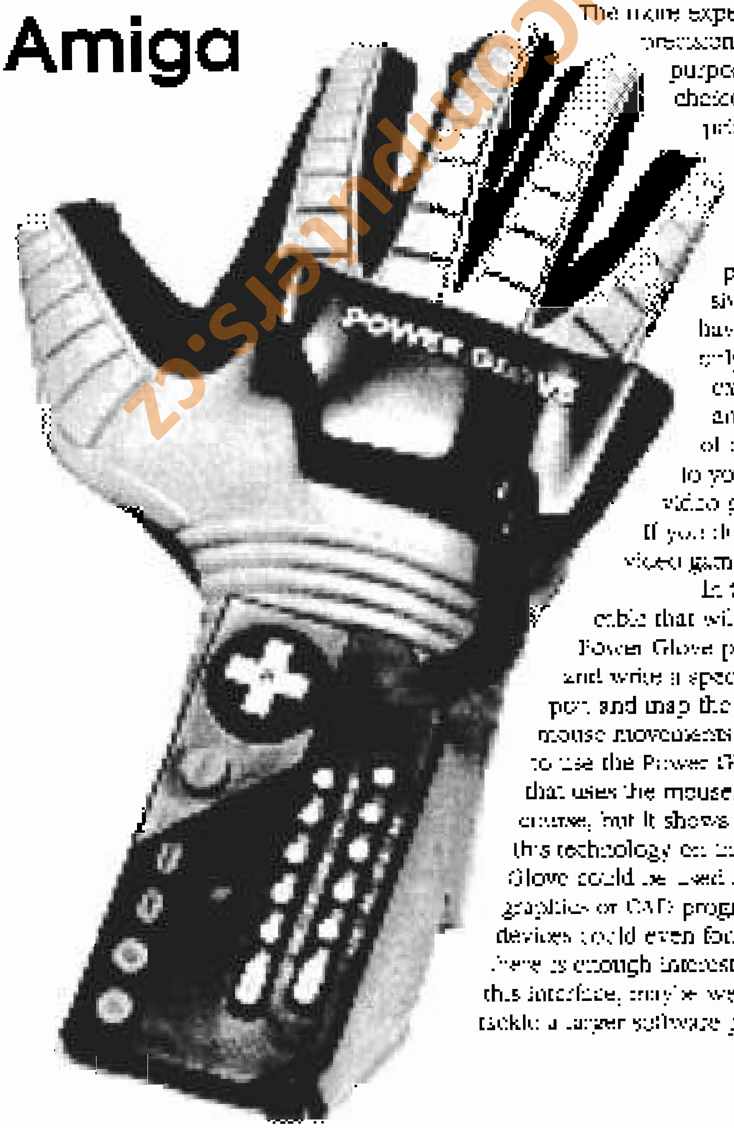
Adapting Mattel's Power Glove to the Amiga

By Paul King and Mike Cargal

bix pking, mcargal

The interface between human and computer has changed very little since the earliest generations of computers, but today there are some new devices coming out that promise to make the task of translating ideas into bits and bytes much more natural. Since its introduction, the Amiga has been one of the most interactive and intuitive computers around, and hardware is obtainable that extends its abilities enough to compete with some much more expensive equipment. It is almost laughable, then, that a popular video game machine has a readily available peripheral that can come closer to natural input than any except the most expensive devices around, while the Amiga and other computers are left out in the cold. That device is Mattel's Power Glove.

While the VPL DataGlove and the Dexterous Hand Master from Exos are both available for computer use, each costs more than your above-average new car. The Power Glove, on the other hand (couldn't resist), can be found for $70-$100 at anywhere from Toys-R-Us to Wal-Mart. The Power Glove has an 8 bit processor that watches a special sensor in each of the fingers and the thumb of the glove, and takes care of tracking the glove by using an ultrasonic ranging system similar to that used in today's Polaroid cameras. Basically, it can inform its host about hand and finger movement with astonishing accuracy. The more expensive devices have greater precision and resolution, but for our purposes the Power Glove is a logical choice, particularly considering its price tag.



So, we have the makings of a very interesting project: inexpensive, readily available, easy to implement, and lots of potential. I guess that "inexpensive" is relative, but if you already have access to a Power Glove it will only take enough money to buy an extension cable and a connector, and if you don't, then the expense of buying one can easily be justified to your spouse if you have a Nintendo video game, especially if you have kids. If you don't have kids or a Nintendo video game then you're on your own. Beg.

In this project we will construct a cable that will interface the peculiar Nintendo/Power Glove plug to the Amiga joystick port and write a special program that will read the port and map the glove's movements to regular mouse movements for Intuition. This will allow you to use the Power Glove with almost any program that uses the mouse. This is of limited usefulness, of course, but it shows a little of what is possible using this technology on the Amiga. Theoretically, the Power Glove could be used as a method of input for 3-D graphics or CAD programs, or in conjunction with other devices could even form the basis of a virtual reality. If there is enough interest in exploring the possibilities of this interface, maybe we could go more in-depth and tackle a larger software project at a later date. For now, let's get the cable built and the Amiga and the Power Glove talking.

When first considering this project, it seemed that the most obvious place to interface the Power Glove to the Amiga was the joystick port (the second port, beside where the mouse plugs in), if it could be done. The glove requires +5 volts, a ground, one input line, and two output lines, one for a reset and one for a clock. As it happens, the power and ground are available on the joystick port (pins 7 & 8 respectively), as are a number of input pins. The only possible problem was outputs.

For normal mouse and joystick use, the Amiga doesn't need any of the pins on the mouse/joystick ports to be used for output. However, there are two special pins that are normally used to read analog joysticks that are software-switchable to be outputs (pins 5 & 9). Because of the way they are used in reading analog joysticks, these two pins are hooked to large capacitors. The resistance in the analog joystick regulates how long it takes for the charge in the capacitors to build up, and by timing that buildup the Amiga is able to determine the joystick's position. Because these pins are hooked to capacitors, there may be a lag of up to 300 microseconds for output to reach the port. Fortunately, one of the outputs we need is not terribly time-critical the reset line. We can use one of these pins for it.

In addition to these two output pins, the port also has another software switchable pin that can be either input or output (pin 6). Usually it is connected to the left mouse button, or to the fire button of a joystick. Since this pin is not hooked up to a capacitor and therefore doesn't have a signal delay, this pin can be used for the clock line (which needs to be as fast as possible). If we choose pin 4 for the data input line, then we can use pins 4-8, with all other pins unused, which makes for a nice, uncluttered interface specification. Looks like the joystick port has everything we need. Too bad other computers don't have such versatile hardware!

This is what you will need to adapt the Power Glove to the Amiga joystick port.

(1)Extension cable for the Nintendo Game controller
Curtis makes Super Extendo, part #NC-3 available at
Wal-Mart for under $10
(1)Nine position female D-subminiature connector
Radio Shack Cat. # 276-1538 for $1.19
(1)D-subminiature connector hood
Radio Shack Cat. # 276-1539B for $.79

Start by cutting off the end of the cable that would normally go to the game console (the smaller plug), then carefully strip off about an inch and a half of the outside layer of plastic to expose the wires inside. Once you do that, strip about a quarter inch off each of the exposed wires. The colors of the wires may vary from cable to cable, so you need to use a multimeter to check which wire leads to each pin in the plug at the other end of the cable. Write it all down.

The connector that is listed here does not require any soldering. The connector package contains a strip that holds all of the individual metal contacts. One at a time, use a pair of needle-nose pliers to crimp one of these contacts onto each wire, then carefully bend the contact back and forth with the pliers to break it away from the metal strip. The pin numbers for the connector are embossed onto the plug itself, so once one of the contacts has been crimped onto all of the wires, match up each wire in the cable with the appropriate position on the connector and push its contact into that hole. You may need to use a toothpick or something similar to push them firmly into place. If you are using a solder type connector, just make sure you solder each wire in the correct position and are careful not to bridge solder between two pins. Figure one shows the proper connections.

After completing all of the connections, attach the hood around the connector to prevent pulling the wires loose. If you own an Amiga 1000 you may need to modify the plastic hood a bit so the connector can be pushed into the port all the way, to ensure proper connection. Once you have finished, use a multimeter to recheck all of the connections to make sure everything is in the right place. While this cable is perfectly safe to the computer and the glove, any wires out of place could cause damage to either or both. CHECK YOUR WORK! If you want to try out this interface but are not a hardware type, you can find ordering information at the end of this article. That completes the hardware part of this project (pretty simple, huh?).

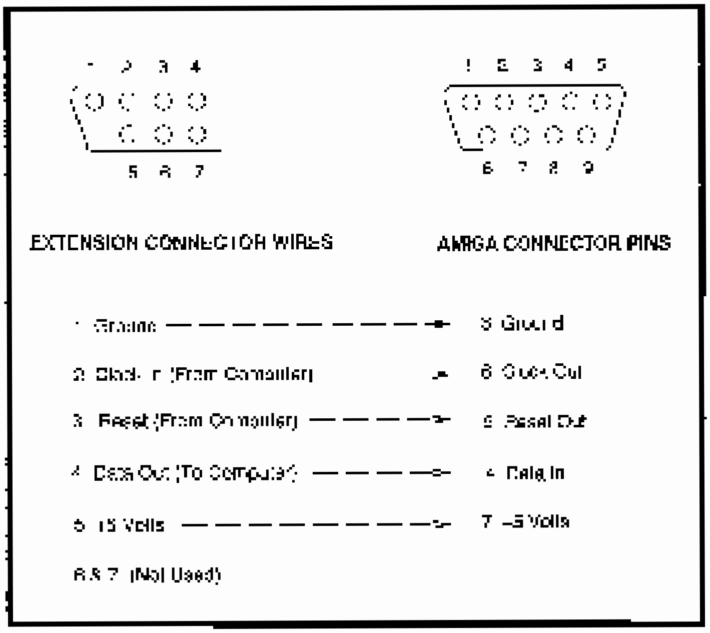
The Power Glove itself handles the processing of its hardware into binary information. It has a set of built-in programs that convert keypad presses and hand and finger movements into a stream of data bits. The default built-in Power Glove program is used in this project, which allows us to use up, down, left, right, thumb, forefinger, and the Start and Select buttons on the keypad. For emulating a mouse, the main ones we are interested in are the directional movements for pointer positioning, and the forefinger and thumb, which we are used for mouse button events. The keypad buttons are used for changing the configuration of the program on the fly.

The hardware interface we have constructed makes all of the physical connections necessary to communicate with the Power Glove through the Amiga's joystick port. Now all we need is software that will actually talk with the glove through the Amiga hardware and convert the data we receive from binary form into mouse events. We call this program PowerMouse.

The Power Glove uses a form of serial communication; it sends a stream of data bits to its host, one bit at a time. Since the glove has no set clock (baud) rate, in order to receive data the host must "poll" the glove: it must tell the glove it is ready to receive the next bit by sending it a signal. The first time this is done, the Amiga sends a 'reset' signal (pin 5), indicating to the glove that it is ready to receive the first bit of data, which the glove then sends. The Amiga then reads its input pin (pin 4) for this bit. For each of the remaining bits (2-8), the Amiga first sends a 'clock' signal (pin 6), indicating to the glove to send the next bit in the data stream. The Amiga then reads its input pin for these bits, sending a clock signal then reading the input pin for each one. After all eight bits have been read, the process is repeated. Each bit represents the state of one of the actions the glove is monitoring.

The PowerMouse program works by installing a background task which polls the joystick port and creates input.device events to simulate mouse movement. The main process opens a control window and processes IDCMP messages in standard Amiga fashion. However, when the window is hidden the main task goes to sleep by "Wait"ing on signals,

Figure one



either from the background task, indicating that the 'SELECT' button was pressed or the background task failed, or from the operating system, letting it know that the process received a break signal.

For the budding Modula-2 programmer, the PowerMouse source includes examples of using the the input.device to feed events to the operating system, using the timer.device to delay in a system friendly manner, starting up background tasks and communicating between tasks using signals, opening windows which function correctly regardless of the system font, and using the joystick port for digital input and output. If you are interested in learning how the PowerMouse program works, please read through the source; it is well commented and should answer any questions you have. We used M2Sprint but tried to be as non-compiler specific as possible. You should be able to make this code work with any dependable Modula-2 compiler with little modification.

Start the program by typing 'run PowerMouse' at the CLI prompt or by double-clicking on its icon. The program starts by bringing up a configuration window which has a number of gadgets that let you control how the program operates. The first thing to do after running the program is turn off the rapid fire feature on the glove ('A OFF' and 'B OFF'), and press 'SELECT' to activate the Power Glove. Now make a fist a few times to calibrate the glove's sensors to the size of your fist. Finally, press the 'START' button to enable the glove.

Once the program is running, you can get rid of the configuration window by clicking on the Hide gadget or by pushing the 'SELECT' button on the glove's keypad. Pressing the 'SELECT' button at any time will toggle the configuration window, from the front to hidden and back. Once the window has been hidden, if for some reason the glove is not responding properly, the only way to kill the PowerMouse program is to send it a control C using the break command from a CLI (use the status command to find the right process number), so make sure the glove is working before hiding the window.

The Enabled gadget toggles glove response. Switching Enabled on (highlighted) lets the glove function normally, and turning it off effectively switches off the glove while letting the program remain running. The Enable function can also be toggled by using the 'START' button on the glove's keypad (it works even if the configuration window is hidden). To stop the program completely click on the Kill gadget.

The Rate slider sets how often the Amiga polls the glove. The sensitivity of the glove is directly proportional to rate setting, while the effect on the system is inversely proportional. The default value should work fine with most machines, but if you have an accelerator card installed or happen to be lucky enough to own a 3000, you may want to set RATE to a lower number, particularly if you are getting erratic response.

The Speed slider sets the number of pixels the pointer moves for each movement event generated by the glove. A lower number is much easier to control, but the pointer takes longer to get where you want it to go. The Accelerate gadget toggles a pointer speed increase mechanism on or off; it is on by default. Acceleration kicks in if you move constantly in one direction, and allows you to move quickly from one side of the screen to the other without losing the ability to do more detailed movement when necessary. These two gadgets should be used together to make PowerMouse respond in a manner which you find comfortable.

If you have any problems with erratic responses that are not affected by these configuration settings, check to make sure the rapid fire and *slow motion* options on the Power Glove are turned off (buttons 5, 7, and 9). If you don't get any response at all, first press 'SELECT' on the Power Glove, and if still nothing then check your cable. It may be wired incorrectly or you may have a bad connection. If problems persist, try a higher quality connector (this would require soldering).

By the way, this interface will also work with other Nintendo game controllers which have more immediate potential than the Power Glove. You can use one of those inexpensive infrared controllers ($20 at Wal-Mart) and have the (almost) equivalent of a cordless mouse, except you don't need a flat surface to use it! This is the best way we have found yet for doing presentations: stand back in the middle of the room and still have complete control of your program.

Also included on the AC's TECH disk, is a version which has a few extra features. For instance, it is configurable for keyboard events as well as mouse events, which allows you to use the glove with some games that use the keyboard for input. You can use it to map out different keystroke sequences for each button or direction of movement, which makes doing presentations or sitting back and reading through messages online a breeze with the cordless controller. —Ed]

Can I Buy the Interface?

The pre-assembled hardware interface and a distribution disk containing the latest version of the PowerMouse program and its source are available for your convenience for $25. If you order, we will also put your name on a mailing list to keep you informed of any future development using this interface for input. Please send a check or money order to the address below. Make sure all checks are drawn on a U.S bank for U.S. funds. Please allow 4 to 6 weeks for delivery.

PowerMouse Distribution
c/o Mike Cargal
7857 Crescent Drive
Columbus, GA 31909

MODULE PowerMouse

```
MODULE PowerMouse;

FROM Glove        IMPORT StartGloveTask, SelectSignal,
                         HaltSignal;
FROM Window       IMPORT ControlWindow;
FROM DOS          IMPORT SIGBreakC, SIGBreakD, SIGBreakE,
                         SIGBreakF;
FROM TasksIMPORT Wait, SignalSet;

(*———————————————————————————————*

   PowerMouse                                               |
                                                            |
    Author:    Mike Cargal     (BIX: mcargal)               |
               with Paul Kline (BIX: pkline)                |
    Date:      Aug. 1990                                    |
    Version  : Original                                     |
                                                            |
   - ---- ------------------------------------------------- |
                                                            |
  First we fire up the background task by calling           |
   StartGloveTask, then we open a control window and        |
   process input from the window.                           |
                                                            |
  When the user closes this window we continue by just      |
   going into a wait for a signal from the glove task       |
   indicating that (1) something has gone wrong and we      |
   need to terminate (HaltSignal) or (2) the user has       |
   pressed the select button to request the control         |
   window (SelectSignal).                                   |
                                                            |
  We will continue this loop until we receive a             |
   HaltSignal or one of the called procedures forces a      |
   HALT.  All cleanup is handled by TERMPROC procedures,    |
   so we can call HALT anytime and anyplace we wish to      |
   exit the program and all resources will be returned      |
   to the system.                                           |
                                                            |
 *                                 *)

VAR
  WaitSignalSet : SignalSet;

BEGIN
  StartGloveTask;
  LOOP
    ControlWindow;
    WaitSignalSet := Wait(SignalSet{SelectSignal,
                                     HaltSignal,
                                     SIGBreakC,
                                     SIGBreakD,
                                     SIGBreakE,
                                     SIGBreakF});
    IF WaitSignalSet *
       SignalSet{HaltSignal,
                 SIGBreakC,
                 SIGBreakD,
                 SIGBreakE,
                 SIGBreakF} # SignalSet{} THEN
      HALT
    END;
  END;

END PowerMouse.
```

DEFINITION MODULE Globals

```
DEFINITION MODULE Globals;

CONST
  MouseRange  = 20;
  SpeedRange  = 7;

VAR
  Mode,
  Speed        : CARDINAL;
  Accelerate,
  Enabled      : BOOLEAN;

END Globals.
```

IMPLEMENTATION MODULE Globals

```
IMPLEMENTATION MODULE Globals;

BEGIN

  Mode       := 11;
  Speed      := 3;
  Accelerate := TRUE;
  Enabled    := FALSE;

END Globals.
```

DEFINITION MODULE Glove

```
DEFINITION MODULE Glove;

VAR
  SelectSignal,
  StartSignal,
  HaltSignal   : CARDINAL;

(*——————————————————————————————*

  These signals are used by the Glove polling task to
   communicate with the main process.

 *——————————————————————————————*)

PROCEDURE StartGloveTask;

(*——————————————————————————————*
 |
 |  Start up background task to poll the glove.
 |
 |  This will signal the main task if it encounters
 |   problems allocating the necessary resources. It
 |   also checks a flag set by the main process in the
 |   TERMPROC to remove itself when the program
 |   terminates.
 |
 *                              *)

END Glove.
```

IMPLEMENTATION MODULE Glove

```
IMPLEMENTATION MODULE Glove;

FROM SYSTEM         IMPORT SHIFT, TERMPROC, LONG, ADR,
                           BYTE, ADDRESS;
FROM PotgoResource  IMPORT PotgoBits, PotgoBitsSet,
                           PotgoName, PotgoBase,
                           AllocPotBits, FreePotBits,
                           WritePotgo;
FROM DOSProcess     IMPORT Delay;
FROM CIAHardware    IMPORT CIA, CIAA, CIAPraBits;
FROM CustomHardware IMPORT custom, Custom;
FROM Resources      IMPORT OpenResource;
FROM Ports          IMPORT [illegible]
FROM Tasks          IMPORT TaskPtr, Task, AddTask, RemTask,
FROM Nodes          IMPORT NodeType;
FROM Memory         IMPORT AllocMem, FreeMem, MemReqSet,
                           MemReqs;
FROM Interrupts     IMPORT Forbid, Permit;
FROM Events         IMPORT MouseEvent, InitEvents,
                           EndEvents;
FROM Globals        IMPORT Mode, Enabled;
FROM Notify         IMPORT Note;
FROM Tasks     IMPORT Signal, SignalSet, CurrentTask,
                           FindTask, AllocSignal, SetSignal,
                           FreeSignal, SetSignals;

(*——————————————————————————————*
    (thanks to Carolyn Scheppner for her article in          |
     AmigaMail "Joystick Port Control". The code is valid    |
     to use [illegible] to translate to Modula-2             |
     [illegible]                                             |
                                                             |
 *                                 *)

CONST
  [illegible]   = PotgoItemSet{OUTRX,DATRX};
  PotRXMask     = PotgoItemSet{OUTRY,DATRY};
  PotRXSet      = PotgoBitSet{OUTRX,
    [illegible]   [illegible]
  StackSize     = 2000;
  Time4         = 0;
  TaskName      = '[illegible] task_level Task';

TYPE
  GloveDirs = (GloveA,
               GloveB,
               GloveSelect,
               GloveStart,
               GloveUp,
               GloveDown,
               GloveLeft,
               GloveRight);

  GloveArray = ARRAY [MIN(GloveDirs)..MAX(GloveDirs)]
               OF BOOLEAN;

  MyTaskRec  = RECORD
                 task  : Task;
                 stack : ARRAY [1..StackSize-1] OF BYTE;
               END;

VAR
  State           : GloveArray;
  MasterProcess   : TaskPtr;
  CurPotBits      : PotgoBitSet;
  MyTask          : POINTER TO MyTaskRec;
  SelectButton,
  StartButton,
  Waiting,
  WaitOK          : BOOLEAN;

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

PROCEDURE StrobeHigh();

(*                                  .
                                                             :
    Sends a [illegible] pulse to pin 5 on the joystick port. :
    This resets the glove for its next [illegible] of 8 bits. .

 *                                  *)

BEGIN
    WritePotgo(PotRXSet,PotRXMask);   (* Set Pin 5 high *)
    WritePotgo(PotRXClr,PotRXMask);   (* Set Pin 5 low *)
END StrobeHigh;

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

PROCEDURE ClockPulse();

 (*-----------------------------------*
  |
  |  Sends a [illegible] pulse to pin 6 of the joystick port.
  |  This signals the glove to [illegible] [illegible] the next
  |   bit [illegible].
  |
  *                                   *)

BEGIN
    EXCL(CIAA^.ciapra,CIAAGamePort1);   (* Set Pin 6 low *)
    INCL(CIAA^.ciapra,CIAAGamePort1);   (* Set Pin 6 high *)
END ClockPulse;

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

PROCEDURE ReadGlove();

 (*                                   *
  |
  |  Loop through, reading each of the data bits from the
  |    glove [illegible] at a time on pin 9 of the joystick
  |    port. After 8 bits are read the data is
  |    interpreted and the appropriate action is taken.

  *          -----------------------*)

VAR
  i   : GloveDirs;
BEGIN

  FOR i := MIN(GloveDirs) TO MAX(GloveDirs) DO
    State[i] := (Pin9 IN BITSET(CIAA^.[illegible]));
    ClockPulse();
  END;

  IF Enabled THEN
    MouseEvent(State[GloveLeft],
               State[GloveRight],
               State[GloveUp],
               State[GloveDown],
               State[GloveB],
               State[GloveA]);
  END;

  IF State[GloveSelect] THEN
    IF NOT SelectButton THEN
      Signal(MasterProcess,SignalSet{SelectSignal});
      SelectButton := TRUE
    END
  ELSE
    SelectButton := FALSE
  END;

  IF State[GloveStart] THEN
    IF NOT StartButton THEN
      Enabled := NOT Enabled;
      Signal(MasterProcess,SignalSet{StartSignal});
      StartButton := TRUE;
    END
  ELSE
    StartButton := FALSE
  END;

  [illegible] := [illegible] * 2500;

  StrobeGlove;

END ReadGlove;

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

PROCEDURE GloveTask();
BEGIN
  IF InitEvents() AND InitPalm() THEN
    StrobeReset;
    REPEAT
      ReadGlove
    UNTIL Waiting;
    EndEvents;
    EndPalm;
  ELSE
    Signal(MasterProcess,SignalSet{HaltSignal});
  END;
  Forbid;
    WaitOK := TRUE;
    RemTask(CurrentTask());
END GloveTask;

(* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *)

PROCEDURE StartGloveTask;

BEGIN

  PotgoBase := OpenResource(ADR(PotgoName));
  IF PotgoBase = NIL THEN
    WriteS("Can't open potgo resource");
    HALT;
  END;

  (* Allocate Pin 5 for our use *)
  CurPotBits := AllocPotBits(CurBits);
  IF CurPotBits # OurBits THEN
    WriteS("Can't allocate potbits");
    HALT;
  END;

  (* Set Pin 6 to output *)
  INCL(CIAA^.ciaddra,CIAAGamePort1);

  MasterProcess := FindTask(CurrentTask);
  SelectSignal  := AllocSignal(AnySignal);
  StartSignal   := AllocSignal(AnySignal);
  HaltSignal    := AllocSignal(AnySignal);
  IF (SelectSignal = CARDINAL(NoSignals)) OR
     (StartSignal    = CARDINAL(NoSignals)) OR
     (HaltSignal     = CARDINAL(NoSignals)) THEN
     WriteS("Could not get signals");
     HALT
  END;

  MyTask := AllocMem(SIZE(MyTaskRec),
                     MemReqSet{MemClear,MemPublic});
  IF MyTask = NIL THEN
    WriteS("Could not allocate glove task");
    HALT
  END;
  WaitOK := FALSE;
  WITH MyTask^.task DO
    tcSPLower := ADR(MyTask^.stack);
    tcSPUpper := tcSPLower + ADDRESS(StackSize);
    tcSPReg   := tcSPUpper;
    WITH taskmem DO
      lnType := NTTASK;
      lnPri  := BYTE(0);
      lnName := ADR(TaskName);
    END;
  END;
  AddTask(ADR(MyTask^.task),ADR(GloveTask),NIL);
END StartGloveTask;

(*-----------------------------------------------------------*)

PROCEDURE CleanUp();
BEGIN

  IF MyTask # NIL THEN
    Halting := TRUE;
    REPEAT
      Delay(10);
    UNTIL Halted;
    FreeMem(MyTask,SIZE(MyTaskDesc));
    MyTask := NIL;
  END;

  IF SelectSignal # CARDINAL(NoSignals) THEN
    FreeSignal(SelectSignal);
    SelectSignal := CARDINAL(NoSignals)
  END;
  IF StartSignal  # CARDINAL(NoSignals) THEN
    FreeSignal(StartSignal);
    StartSignal := CARDINAL(NoSignals)
  END;
  IF HaltSignal   # CARDINAL(NoSignals) THEN
    FreeSignal(HaltSignal);
    HaltSignal := CARDINAL(NoSignals)
  END;

  IF CurSetBits # DataBitSet{} THEN
    WriteDOut(DataBitSet{});    (* Set Pin 5 back low *)
    CurSetBits := DataBitSet{}
  END;
  SetDir(DataBitSet{});    (* Set Pin 1 low *)
  FreeMiscResource(MRParallelPort);  (* Set Pin 8 to input *)
END CleanUp;

(*-----------------------------------------------------------*)

BEGIN
  IF FindTask(ADR(TaskName)) # NIL THEN
    WriteString("Glove/Mouse is already running!");
    HALT
  END;
  SelectButton  := FALSE;
  StartButton   := FALSE;
  Halting       := FALSE;
  Halted        := TRUE;
  MyTask        := NIL;

  TERMPROC(CleanUp);

END Glove.
```

DEFINITION MODULE Events

```
DEFINITION MODULE Events;

PROCEDURE WriteEvent(left,
                     right,
                     up,
                     down,
                     button,
                     select : BOOLEAN);

(*                                            *)
                                                    |
     Send an InputEvent to the input.device to simulate
       mouse input.                                   |
                                                    |
     We just need to know the current state of the glove
      signals and will track changes (button up or down,
      acceleration etc.) internally.                 |
                                                    |
 *--------------------------------------*)

PROCEDURE InitEvents () : BOOLEAN;

(*--------------------------------------*)
                                                    |
     Allocate the resources necessary to send InputEvents
      to the input.device.                           |
                                                    |
     This proc must be called from within the task which
      will be calling WriteEvent.  This is why it is not
      just part of the startup for this module.      |
                                                    |
 *--------------------------------------*)

PROCEDURE EndEvents ();

(*--------------------------------------*)

    Free the resources allocated by InitEvents.

    This proc should also be called from within the task
     which has been calling WriteEvents, and cannot
     therefore be set up as a TERMPROC.

 *                                      *)

END Events.
```

IMPLEMENTATION MODULE Events

```
IMPLEMENTATION MODULE Events;

FROM SYSTEM          IMPORT BYTE, ADR, SHIFT, ADDRESS,
                            REGISTER, SETREG;
FROM Devices         IMPORT OpenDevice, CloseDevice;
FROM IO              IMPORT IOStdReq, IOStdReqPtr, DoIO,
                            IOFlagSet, SendIO, WaitIO,
                            AbortIO;
FROM IOUtils         IMPORT CreateStdIO, DeleteStdIO;
FROM Ports           IMPORT MsgPortPtr, MsgPort;
FROM PortUtils       IMPORT CreatePort, DeletePort;
FROM InputEvents     IMPORT InputEvent, IECodeSet,
                            IEQualifierSet, IEClass,
                            InputEventPtr, IECodeLButton,
                            IECodeRButton, IECodeNoButton,
                            IEClassRawMouse;
FROM InputDevice     IMPORT INDAddEvent, InputDeviceName;
FROM Terminal        IMPORT WriteString;
FROM Kernel          IMPORT Chan;
FROM Globals         IMPORT Speed, Accelerate;

CONST
  AccelerationThreshold = 3;

TYPE
  Direction = (Up, Down, Left, Right);

VAR
  Outstanding,
  InputDevIsOpen        : BOOLEAN;
  Req                   : IOStdReqPtr;
  IE                    : InputEvent;
  MyPort                : MsgPortPtr;
  InputReq              : IOStdReqPtr;
   LeftButton,
  RightButton           : BOOLEAN;
  i                     : CARDINAL;
  Spd                   : ARRAY
                          [MIN(Direction)..MAX(Direction)]
                          OF INTEGER;
  Times                 : ARRAY
                          [MIN(Direction)..MAX(Direction)]
                          OF CARDINAL;
  Going                 : ARRAY
                          [MIN(Direction)..MAX(Direction)]
                          OF BOOLEAN;

(*-----------------------------------------------------------*)

PROCEDURE Accel (VAR move : INTEGER;
                   active : BOOLEAN;
                      dir : Direction);

(*--------------------------------------*)

    Increases mouse speed when a direction is selected
     continuously.

 *                                      *)

BEGIN
  IF active THEN
    IF Going[dir] THEN
      INC(Times[dir]);
      IF Times[dir] > AccelerationThreshold THEN
        INC(Spd[dir],Speed);
        Times[dir] := 0
      END
    ELSE
      Going[dir] := TRUE;
      Times[dir] := 0;
      Spd[dir]   := Speed;
    END;
    IF (dir = Up) OR (dir = Left) THEN
      DEC(move,Spd[dir])
    ELSE
      INC(move,Spd[dir])
    END
  ELSE
```

```
      Paint[dir] := FALSE
    END;
END Button;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

PROCEDURE MoveMouse (left,
                     right,
                     up,
                     down,
                     lbutton,
                     rbutton : BOOLEAN);

VAR
  Doit : BOOLEAN;
  Acc  : INTEGER;

BEGIN

  IF (left AND right) OR (up AND down) THEN
    RETURN
  END;

  IF Outstanding THEN
    msg := WaitIO(Req)
  END;

  Doit := FALSE;
  WITH IE DO
    ieCode       := IECodeNoButton;
    ieQualifier  := InputEventQualifierSet{ieqRelativeMouse};
    ieClass      := IEClassRawMouse;
    ieSubClass   := BYTE(0);
    ieX := 0;
    ieY := 0;

    IF lbutton THEN
      INCL(ieQualifier,IEQualifierLeftButton);
      IF NOT LeftButton THEN
        Doit       := TRUE;
        ieCode     := IECodeLButton;
        LeftButton := TRUE;
      ELSE
      END;
    ELSE
      IF LeftButton THEN
        Doit       := TRUE;
        ieCode     := IECodeLButton+IECodeUpPrefix;
        LeftButton := FALSE;
      END
    END;

    IF rbutton THEN
      INCL(ieQualifier,IEQualifierRButton);
      IF NOT RightButton THEN
        Doit        := TRUE;
        ieCode      := IECodeRButton;
        RightButton := TRUE;
      END;
    ELSE
      IF RightButton THEN
        Doit        := TRUE;
        ieCode      := IECodeRButton+IECodeUpPrefix;
        RightButton := FALSE;
      END
    END;

    IF Accelerate THEN
      Accel(ieX,left, Left);
      Accel(ieX,right,Right);
      Accel(ieY,up,   Up);
      Accel(ieY,down, Down);
    ELSE
      IF left  THEN DEC(ieX,Speed) END;
      IF right THEN INC(ieX,Speed) END;
      IF up    THEN DEC(ieY,Speed) END;
      IF down  THEN INC(ieY,Speed) END;
    END;

    IF (ieX # 0) OR (ieY # 0) THEN
      Doit := TRUE;
    END;

  END;

  IF Doit THEN
    SendIO(Req);
    Outstanding := TRUE;
  END;

END MoveMouse;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

PROCEDURE InitDevice() : BOOLEAN;

BEGIN
  MyPort := CreatePort(NIL, 0);
  Req    := CreateStdIO(MyPort);
  IF Req = NIL THEN
    RETURN FALSE
  END;

  WITH Req^ DO
    ioCommand := INDWriteEvent;
    ioLength  := SIZE(InputEvent);
    ioData    := ADR(IE);
  END;

  WITH IE DO
    ieNextEvent         := NIL;
    ieEventAddress      := NIL;
    ieTimeStamp.tvSecs  := 0;
    ieTimeStamp.tvMicro := 0;
  END;

  IF OpenDevice(ADR(InputDeviceName),0,Req,IOStdReqFlagSet{})
     # 0 THEN
    RETURN FALSE
  END;
  InputDevIsOpen := TRUE;

  RETURN TRUE

END InitDevice;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

PROCEDURE EndEvents;

VAR
  msg : INTEGER;
BEGIN
  IF Outstanding THEN
    msg := WaitIO(Req);
    Outstanding := FALSE;
  END;

  IF InputDevIsOpen THEN
    CloseDevice(Req);
    InputDevIsOpen := FALSE;
  END;

  IF Req # NIL THEN
    DeleteStdIO(Req);
    Req := NIL
  END;

  IF MyPort # NIL THEN
    DeletePort(MyPort);
    MyPort := NIL
  END;

END EndEvents;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

BEGIN
  MyPort          := NIL;
  Req             := NIL;
  InputDevIsOpen  := FALSE;
  Outstanding     := FALSE;
  RightButton     := FALSE;
  LeftButton      := FALSE;

END Events.
```

DEFINITION MODULE Window

```
DEFINITION MODULE Window;

PROCEDURE OpenWindow();

(*
 |                                                          |
 | Open a control window, requesting for the system font   |
 |  and falling back to Topaz 8 if the default is too      |
 |  large.                                                  |
 |                                                          |
 | When we open the window we will loop waiting for        |
 |  signals from the keyboard, timer or messages from      |
 |  the window's IDCMP.                                     |
 |                                                          |
 | This routine returns to the caller when the window is   |
 |  closed, and this WILL happen when the user             |
 |  clicks on the CLOSE gadget.                             |
 |                                                          |
 *)

END Window.
```

IMPLEMENTATION MODULE Window

```
IMPLEMENTATION MODULE Window;

FROM SYSTEM       IMPORT ADR, BYTE, [illegible], ADDRESS, LONG,
                         TSIZE, [illegible];
FROM Intuition    IMPORT Screen, [illegible], Window,
                         [illegible], CloseWindow, ReportMouse,
                         GetScreenData, ModifyProp,
                         [illegible], [illegible], [illegible],
                         [illegible], RefreshGadgets,
                         [illegible], [illegible], Gadget,
                         GadgetPtr, [illegible], PropGadget,
                         [illegible], [illegible],
                         [illegible], [illegible],
                         [illegible], [illegible],
                         IntuiMessagePtr, IDCMPFlags,
                         IDCMPFlagSet, [illegible],
                         [illegible], [illegible], PropInfo,
                         [illegible], [illegible], PropFlags,
                         WindowPtr, [illegible],
                         WBenchScreen, [illegible],
                         [illegible], [illegible], [illegible],
                         Image, ImagePtr, AddGadget,
                         RemoveGadget, [illegible],
                         WBenchToFront;
FROM Memory       IMPORT [illegible], [illegible],
FROM Text         IMPORT TextAttr, TextAttrPtr, TextFont,
                         TextFontPtr, OpenFont, CloseFont,
                         FontFlags, [illegible], [illegible];
FROM Ports        IMPORT GetMsg, ReplyMsg, WaitPort;
FROM [illegible]  IMPORT [illegible], [illegible];
FROM [illegible]  IMPORT [illegible], [illegible];
FROM [illegible]  IMPORT Real, Trunc;
FROM Conversions  IMPORT [illegible], [illegible];
FROM [illegible]  IMPORT [illegible];
FROM [illegible]  IMPORT Rate, RateRange, Speed, SpeedRange,
                         Accelerate, Enabled;
FROM [illegible]  IMPORT [illegible], [illegible], [illegible];
FROM Tasks        IMPORT SignalSet, [illegible];

CONST
  Default  = FALSE;
  FullSize = TRUE;

VAR
  wp                  : WindowPtr;
  RateGadget,
  SpeedGadget,
  AccelerateGadget,
  EnabledGadget,
  HideGadget,
  KillGadget          : GadgetPtr;
  [illegible]         : [illegible];
  DefFont             : TextFontPtr;
  [illegible],
  [illegible]         : [illegible];
  DefTextAttr         : TextAttr;
  ItemSelect          : BOOLEAN;

(*-----------------------------------------------*)

PROCEDURE ScaleAndOpenWindow (FallingBack : BOOLEAN) :
                                              BOOLEAN;

(*------------------------------------*
 |                                                  |
 | Determine the size window and gadgets necessary to |
 |  accomodate the default system font (FallingBack = |
 |  FALSE).  If the default font is too large to    |
 |  accomodate, this routine will call itself       |
 |  recursively specifying that it should attempt to fall |
 |  back to Topaz 8 (FallingBack = TRUE)            |
 |
 | It returns TRUE to indicate that it has successfully
 |  opened the control window, or FALSE to indicate that
 |  it was unable, for some reason, to open even a
 |  window with the fallback font.
 |
 *-------------------------------          *)

CONST
  AccelerateText = "Accelerate";
  EnabledText      "Enabled";
  HideText         "Hide";
  KillText       = "Kill";
  RateText       = "Rate  ";
  SpeedText      = "Speed ";

VAR
  DefRect       : RECORD
                    x, y : INTEGER;
                  END;
  sp            : ScreenPtr;
  nw            : NewWindowPtr;
  [illegible]   : [illegible];
  [illegible], [illegible],
  [illegible]   : INTEGER;
```

```
FirstGadget : BOOLEAN;

(*------------------------------------------*)

PROCEDURE [illegible] : INTEGER) : INTEGER;
BEGIN
  IF a > b THEN
    RETURN a
  END;
  RETURN b
END [illegible];

(*------------------------------------------*)

PROCEDURE [illegible] : INTEGER;
BEGIN
  IF a > b THEN
    RETURN a
  ELSE
    RETURN b
END greatest;

(*------------------------------------------*)

PROCEDURE [illegible] : STRING);

(*-------------------------------*
 |
 |  This routine will calculate the size [illegible]
 |   to enclose "str". [illegible]
 |   [illegible] is greater than
 |   the default as far.
 |
 |  After this routine has been called [illegible]
 |   DefRect will contain [illegible]
 |   [illegible] string "str" for the default.
 |
 *-                            *)

VAR
  it : IntuiText;
  [illegible]
  len : INTEGER;

BEGIN
  (* Height of font plus 2 pixels top & bottom. *)
  ry := DefFont^.[illegible] + 4;

  WITH it DO
    ITextFont := ADR(DefTextAttr);
    IText     := str;
  END;

  (* Width of text plus 4 pixels left and right *)
  rx := IntuiTextLength(ADR(it)) + 8;

  WITH DefRect DO
    x := Greatest(x, rx);
    y := Greatest(y, ry);
  END
END [illegible];

(*------------------------------------------*)

 [illegible] x, y           INTEGER;
                  label     : STRING;
                  toggle    : BOOLEAN;
                  selected    BOOLEAN;
                  nextgad     GadgetPtr) : GadgetPtr;

(*-------                              *

   Allocate a boolean gadget [illegible]
     [illegible]

   'x' and 'y' are the rectangle coordinates.

   'label' is the text to use for the button.

   'toggle' indicates that the gadget should be
     ToggleSelect if TRUE.

   'selected' indicates that the gadget
     should come up in a selected state.

   'nextgad' is a pointer to the gadget with which this
     one should be linked.  NIL is valid only for the
     first gadget created; NIL [illegible] received
     thereafter is indicates that an error has occured
 |   in earlier allocations, so we just return NIL to
 |   pass the error condition on.
 |
 *--------------------------------      *)

TYPE
  Button = RECORD
             gad   : Gadget;
             it    : IntuiText;
             [illegible] : [illegible];
             [illegible] : ARRAY [0..4],[0..1] OF INTEGER;
           END;
  ButtonPtr = POINTER TO Button;

VAR
  bp : ButtonPtr;

BEGIN
  IF nextgad = NIL THEN
    IF FirstGadget THEN
      FirstGadget := FALSE
    ELSE
      RETURN NIL
    END
  END;

  bp := AllocRemember(MainScreen,
                      SIZE(Button),
                      MemReqSet{MemPublic,MemClear});
  IF bp = NIL THEN
    RETURN NIL
  END;

  WITH bp^ DO
    WITH gad DO
      NextGadget     := nextgad;
      LeftEdge       := VAL(INTEGER,win^.BorderLeft) +
                        (x * DefRect.x) + 4;
      TopEdge        := VAL(INTEGER,win^.BorderTop) +
                        VAL(INTEGER,win^.RPort^.TxHeight) +
                        (y * DefRect.y) + 2;
      Width          := DefRect.x - 10;
      Height         := DefRect.y - 11;
      Flags          := GadgHComp;
      IF selected THEN
        INCL(Flags,Selected)
      END;
      Activation     := ActivationFlagSet{RelVerify};
      IF toggle THEN
        INCL(Activation,ToggleSelect);
      END;
      GadgetType     := BoolGadget;
      GadgetRender   := ADR(bord);
      GadgetText     := ADR(itxt);
    END;

    WITH bord DO
      LeftEdge := 0;
      TopEdge  := 0;
      FrontPen := DETPEN(1);
      DrawMode := Jam1;
      Count    := SIZE(...);
      XY       := ADR(coord);
    END;

    coord[0,0] := 0;               coord[0,1] := 0;
    coord[1,0] := gad.Width - 1;   coord[1,1] := 0;
    coord[2,0] := gad.Width;       coord[2,1] := gad.Height;
    coord[3,0] := gad.Width;       coord[3,1] := gad.Height;
    coord[4,0] := gad.Width;       coord[4,1] := 0;
    coord[5,0] := gad.Width - 1;   coord[5,1] := gad.Height;
    coord[6,0] := -1;              coord[6,1] := gad.Height;
    coord[7,0] := 0;               coord[7,1] := 0;
    coord[8,0] := 0;               coord[8,1] := 0;
    coord[9,0] := 0;               coord[9,1] := gad.Height;

    WITH itxt DO
      FrontPen   := DETPEN(1);
      DrawMode   := Jam1;
      TopEdge    := 3;
      ITextFont  := ADR(DefaultFont);
      IText      := label;
    END;
    itxt.LeftEdge  := (gad.Width - ...) DIV 2 -
                      (INTEGER(TextLength(...)))
                      DIV 2;

    RETURN ADR(gad);
  END;

END AddButton;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

PROCEDURE AddSlider(x,y      : INTEGER;
                    value,
                    range    : CARDINAL;
                      win    : WindowPtr) : GadgetPtr;

(*                               \
                                                        |
   Adds a proportional gadget in the same manner as     |
     AddButton adds a boolean gadget.                   |
                                                        |
   'value' indicates the current value of the variable  |
     to be adjusted by this proportional gadget.        |
 .                                                      |
 . 'range' indicates the range of possible values for   |
     the variable to be adjusted by this proportional   |
 |   gadget.                                            |
 |                                                      |
   'nextgad' (see AddButton)                            |
                                                        |
\ ______________________________________*)

TYPE
  Slider = RECORD
             gad  : Gadget;
             pi   : PropInfo;
             im   : Image
           END;
  SliderPtr = POINTER TO Slider;

VAR
  sp : SliderPtr;
  hp : CARDINAL;
  lc : LONGCARD;

BEGIN
  IF nextgad = NIL THEN
    IF FirstGadget THEN
      FirstGadget := FALSE
    ELSE
      RETURN NIL
    END
  END;

  sp := AllocRemember(MainScreen,
                      SIZE(Slider),
                      MemReqSet{MemPublic,MemClear});
  IF sp = NIL THEN
    RETURN NIL
  END;

  WITH sp^ DO
    WITH gad DO
      NextGadget    := nextgad;
      LeftEdge      := VAL(INTEGER,win^.BorderLeft) +
                       (x * DefRect.x) + 3;
      TopEdge       := VAL(INTEGER,win^.BorderTop) +
                       VAL(INTEGER,win^.RPort^.TxHeight) +
                       (y * DefRect.y) + 4;
      Width         := DefRect.x - 7;
      Height        := DefRect.y - 9;
      Flags         := GadgHComp;
      Activation    := ActivationFlagSet{GadgImmediate,
                                         FollowMouse};
      GadgetType    := PropGadget;
      GadgetRender  := ADR(im);
      GadgetText    := NIL;
      SpecialInfo   := ADR(pi);
    END;

    WITH pi DO
      Flags      := PropFlagSet{AutoKnob,FreeHoriz};
      lc :=  LONGCARD(value) * 0FFFFH DIV
             LONGCARD(range-1);
      HorizPot   := lc;
      VertPot    := 0FFFFH;
      HorizBody  := 0FFFFH DIV range;
      VertBody   := 0FFFFH;
    END;

    RETURN ADR(gad)
  END

END AddSlider;

(*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*)

PROCEDURE AddLabel(gad      : GadgetPtr;
                   x,y      : INTEGER;
                   str      : STRPTR;
                   color,
                   chars    : CARDINAL) : ADDRESS;

(*                               \
                                                        .
   Add an IntuiText label with ... relative ... Passing |
    back an IntuiTextPtr to an IntuiText structure for  |
    updating the value as the slider is manipulated.    |

\_______________________________________*)

TYPE
  Label = RECORD
            it   : IntuiText;
            txt  : ARRAY [0..10] OF CHAR;
          END;
  LabelPtr = POINTER TO Label;

VAR
  lp : LabelPtr;

BEGIN
  lp :=  AllocRemember(MainScreen,
                       SIZE(Label),
                       MemReqSet{MemPublic,MemClear});
  IF lp = NIL THEN
    RETURN NIL
    END;

    WITH gp^ DO
      WITH it DO
        FrontPen   := BYTE(1);
        DrawMode   := Jam1;
        LeftEdge   := VAL(INTEGER,sl^.[illegible])
                      [illegible];
        TopEdge    := VAL(INTEGER,[illegible] -
                      VAL(INTEGER,sl^.TextHeight) +
                      [illegible];
        ITextFont  := ADR(DefTextAttr);
        NextText   := NIL;
      END;

      [illegible](gp^.[illegible],ADR([illegible]),2);

      INC(it.LeftEdge,IntuiTextLength(ADR(it)));

      WITH it DO
        IText      := ADR(buf);
        NextText   := NIL;
      END;
      [illegible](gp^.[illegible],
                        [illegible],
                        [illegible] THEN
        PrintIText(gp^.RPort,ADR(it),0,0);
      END;

      RETURN ADR(it);
    END;

  END AddLabel;

  (*--------------------------------------------------------*)

  PROCEDURE FreeStuff;

  BEGIN
    IF DefFont # NIL THEN
      CloseFont(DefFont);
      DefFont := NIL;
    END;

    IF ScaleParams # NIL THEN
      FreeRemember(ScaleRemember,TRUE);
      ScaleRemember := NIL;
    END;

    IF WindowMsg # NIL THEN
      FreeRemember(RemKey,TRUE);
      ScaleRemember := NIL;
    END;

  END FreeStuff;

  (*--------------------------------------------------------*)

  PROCEDURE TryFallBack () : BOOLEAN;

  BEGIN
    FreeStuff;

    IF FallingBack THEN
      RETURN FALSE
    END;

    RETURN ScaleAndOpenWindow(FallBack);

  END TryFallBack;

  (*--------------------------------------------------------*)

BEGIN
  ScaleRemember := NIL;

  sc := AllocRemember(ScaleRemember,
                      SIZE(ScaleR...),
                      MemReqSet{MemClear});
  IF (sc = NIL) OR
     [illegible](ScaleRemember,
                        ADR(Screen),
                        WBenchScreen,NIL) THEN
    FreeStuff;
    RETURN FALSE
  END;

  IF NOT FallingBack THEN
    Forbid;
      WITH GfxBase^ DO
        DefTextAttr.taName  := taName;
        DefTextAttr.taYSize := taYSize;
        DefTextAttr.taStyle := taStyle;
        DefTextAttr.taFlags := taFlags;
      END;
      DefFont := OpenFont(ADR(DefTextAttr));
    Permit;
  ELSE
    WITH DefTextAttr DO
```

```
      taName  := ADR("topaz.font");
      taYSize := 8;
      taStyle := NormalStyle;
      taFlags := FontFlagSet{ROMFont}
    END;
    DefFont := OpenFont(ADR(DefTextAttr));
  END;
  IF DefFont = NIL THEN
    RETURN TryFallBack()
  END;

  WITH DefRect DO
    x := 0;
    y := 0;
  END;

  ScaleBox(ADR([illegible]));
  ScaleBox(ADR([illegible]));
  ScaleBox(ADR(HideText));
  ScaleBox(ADR(KillText));

  FirstGadget        := NIL;
  RateGadget         := AddSlider(0,1,
                                  Rate.RateRange,
                                  NIL);
  SpeedGadget        := AddSlider(0,2,
                                  Speed.SpeedRange,
                                  RateGadget);
  AccelerateGadget   := AddToggle(1,2,
                                  ADR(AccelerateText),
                                  TRUE,Accelerate,
                                  SpeedGadget);
  EnabledGadget      := AddToggle(1,1,
                                  ADR(EnabledText),
                                  TRUE,Enabled,
                                  AccelerateGadget);
  HideGadget         := AddToggle(1,3,
                                  ADR(HideText),
                                  FALSE,FALSE,
                                  EnabledGadget);
  KillGadget         := AddToggle(1,3,
                                  ADR(KillText),
                                  FALSE,FALSE,
                                  HideGadget);

  IF KillGadget = NIL THEN
    RETURN TryFallBack();
  END;

  nw := AllocRemember(ScaleRemember,
                      SIZE(NewWindow),
                      MemReqSet{MemClear});
  IF nw = NIL THEN
    RETURN TryFallBack();
  END;

  WITH sc^ DO
    WindowX := [illegible]
               VAL(INTEGER,[illegible]) +
               VAL(INTEGER,WBorRight);
    WindowY := [illegible]
               VAL(INTEGER,[illegible]) +
               VAL(INTEGER,WBorTop) +
               VAL(INTEGER,WBorBottom);
  END;

  WITH nw^ DO
    LeftEdge    := [illegible]              [illegible]
                   [illegible]              (WindowX DIV 2));
    TopEdge     := [illegible]              (WindowY DIV 2)
                   [illegible]              [illegible];
    Width       := WindowX;
    Height      := WindowY;
    DetailPen   := BYTE(0);  (* [illegible] *)
    BlockPen    := BYTE(1);
    IDCMPFlags  := IDCMPFlagSet{CloseWindow,
                                GadgetUp,
                                GadgetDown};
    Flags       := WindowFlagSet{WindowDrag,
                                 WindowDepth};
    FirstGadget := KillGadget;
    Title       := ADR("PowerMouse");
    Screen      := NIL;  (* Done by OpenWindow *)
    Type        := WBenchScreen;
  END;

  wp := OpenWindow(nw^);

  IF wp = NIL THEN
    RETURN TryFallBack();
  ELSE
    RateLIT     := AddLabel(wp,0,1,ADR(RateText),Rate,2);
    SpeedLIT    := AddLabel(wp,0,2,ADR(SpeedText),Speed,2);
    FreeRemember(ScaleRemember,TRUE);
    ScaleRemember := NIL;
    RETURN TRUE
  END

END ScaleAndOpenWindow;
```

```
(*------------------------------------------------------------*)

PROCEDURE ProcessSlider(VAR value : CARDINAL;
                            gad   : GadgetPtr;
                            range : CARDINAL;
                            X     : INTEGER);
VAR
  ClrVal : CARDINAL;
  lo     : LONGCARD;

BEGIN
  ClrVal := value;
  lo     := LONGCARD(range-1)
            * LONG(gad^.SpecialInfo^.HorizPot)+1000H)
            DIV MaxPot;
  value  := CARDINAL(lo+1);
  IF value # ClrVal THEN
    it^.FrontPen := BYTE(0);
    PrintIText(wp^.RPort,it,X,0);
    it^.FrontPen := BYTE(1);
    IF ConvNumToStr(it^.IText^,
                    LONG(value),
                    Dec,FALSE,3,' ') THEN
      PrintIText(wp^.RPort,it,X,0);
    END
  END;
END ProcessSlider;

(*------------------------------------------------------------*)

PROCEDURE HandleStartSignal;

VAR
  posn : CARDINAL;

BEGIN

  posn := RemoveGadget(wp,EnabledGadget^);
  IF Enabled THEN
    INCL(EnabledGadget^.Flags,Selected)
  ELSE
    EXCL(EnabledGadget^.Flags,Selected)
  END;
  SetAPen(wp^.RPort,0);
  WITH EnabledGadget^ DO
    RectFill(wp^.RPort,
             LeftEdge,
             TopEdge,
             LeftEdge+Width-1,
             TopEdge+Height-1)
  END;
  posn := AddGadget(wp,EnabledGadget,posn);
  RefreshGList(EnabledGadget,wp,NIL,1)

END HandleStartSignal;

(*------------------------------------------------------------*)

PROCEDURE CleanUp;

BEGIN
  IF wp # NIL THEN
    CloseWindow(wp);
    wp := NIL;
  END;

  IF DefFont # NIL THEN
    CloseFont(DefFont);
    DefFont := NIL;
  END;

  FreeSignal(MainSignal.INT(...));
  MainTask := NIL;
END CleanUp;

(*------------------------------------------------------------*)

PROCEDURE ControlWindow;

VAR
  im            : IntuiMessagePtr;
  cls           : IDCMPFlagSet;
  gad           : GadgetPtr;
  Done          : BOOLEAN;
  MouseMoved    : BOOLEAN;
  WaitSignalSet : SignalSet;

BEGIN
  IF NOT OpenControlWindow() THEN
    Tell("Unable to open Control window!\n");
    EXIT;
  END;

  IF NOT OpenDefFont() THEN
    CleanUp;
    RETURN
  END;
  Done       := FALSE;
  MouseMoved := FALSE;
  REPEAT
    WaitSignalSet := Wait(SignalSet{SelectSignal,
                                    StartSignal,
                                    MainSignal,
                        CARDINAL(wp^.UserPort^.mpSigBit)});
    IF MainSignal IN WaitSignalSet THEN
      HALT
    END;
    IF StartSignal IN WaitSignalSet THEN
      HandleStartSignal;
    END;
    IF SelectSignal IN WaitSignalSet THEN
      IF NOT FirstSelect THEN
        WindowToFront(wp)
      ELSE
        FirstSelect := FALSE
      END
    END;
    im := GetMsg(wp^.UserPort^);
    WHILE (im # NIL) AND (NOT Done) DO
      WITH im^ DO
        cls := Class;
        gad := IAddress
      END;
      ReplyMsg(im);

      IF (GadgetDown IN cls) OR (MouseMove IN cls) THEN
        MouseMoved := TRUE
      END;
      IF GadgetUp IN cls THEN
        IF    gad = KillGadget THEN
          HALT
        ELSIF gad = HideGadget THEN
          Done := TRUE
        ELSIF gad = AccelerateGadget THEN
          Accelerate := NOT Accelerate
        ELSIF gad = EnabledGadget THEN
          Enabled := NOT Enabled
        END;
      END;
      IF NOT Done THEN
        im := GetMsg(wp^.UserPort^)
      END;
    END;
    IF (NOT Done) AND MouseMoved THEN
      ProcessSlider(Rate, RateGadget, RateRange, RateX);
      ProcessSlider(Speed,SpeedGadget,SpeedRange,SpeedX);
      MouseMoved := FALSE;
    END;
  UNTIL Done;
  CleanUp;
END ControlWindow;

(*------------------------------------------------------------*)

BEGIN

  wp          := NIL;
  MainTask    := NIL;
  DefFont     := NIL;
  FirstSelect := TRUE;

  TermProc(CleanUp);

END Window.
```

DEFINITION MODULE Pause

```
DEFINITION MODULE Pause;

PROCEDURE PauseMicros(t : LONGCARD);

(*                              *
 |                                                        |
 |  Use the timer.device to pause for the specified       |
 |   number of micro-seconds.                             |
 |                                                        |
 *                 ------------------*)

PROCEDURE InitPause() : BOOLEAN;

(*                              *
 |                                                        |
 |  Allocate the resources necessary to use the           |
 |   timer.device to pause a task for a specified number  |
 |   of micro seconds.                                    |
 |                                                        |
 |  This code must be called from within the task which   |
 |   will be calling Pause which is why it is not just    |
 |   part of the startup for this MODULE.                 |
 |                                                        |
 |                                                        |
 *------------------------------*)

PROCEDURE EndPause();
```

```
(*
    Free the resources allocated by InitPause.

    This code should also be called even within the task
    which has been [illegible]
    [illegible] set up as a TERMPROC.
*)

END Pause.
```

IMPLEMENTATION MODULE Pause

```
IMPLEMENTATION MODULE Pause;

FROM SYSTEM       IMPORT ADR, BYTE;
FROM Devices      IMPORT OpenDevice, CloseDevice;
FROM IO           IMPORT DoIO, IOStdReq, IOStdReqPtr;
FROM PortsUtil    IMPORT CreatePort, DeletePort;
FROM IOUtil       IMPORT CreateExtIO, DeleteExtIO;
FROM Ports        IMPORT MsgPortPtr;
FROM TimerDevice  IMPORT TimerName, TRAddRequest,
                         UnitMicroHz, TimeVal,
                         TimeValPtr, TimeRequest,
                         TimeRequestPtr;

VAR
  MyTimerMsg  : TimeRequestPtr;
  MyMsgPort   : MsgPortPtr;
  TimerOpen   : BOOLEAN;

(*-----------------------------------------------------------*)

PROCEDURE PauseMicros(micros : LONGCARD);

VAR
  err : INTEGER;

BEGIN
  WITH MyTimerMsg^ DO
    trNode.ioCommand := TRAddRequest;
    trTime.tvSecs    := 0;
    trTime.tvMicro   := LONGINT(micros);
  END;
  err := DoIO(MyTimerMsg);

END PauseMicros;

(*-----------------------------------------------------------*)

PROCEDURE InitPause() : BOOLEAN;
BEGIN

  MyMsgPort := CreatePort(NIL, 0);
  IF MyMsgPort = NIL THEN
    RETURN FALSE
  END;

  MyTimerMsg := CreateExtIO(MyMsgPort, SIZE(TimeRequest));
  IF MyTimerMsg = NIL THEN
    RETURN FALSE
  END;

  IF OpenDevice(ADR(TimerName),
                UnitMicroHz,
                MyTimerMsg,
                LONGBITSET{}) # 0 THEN
    RETURN FALSE
  END;

  TimerOpen := TRUE;
  RETURN TRUE;

END InitPause;

(*-----------------------------------------------------------*)

PROCEDURE EndPause;

BEGIN
  IF TimerOpen THEN
    CloseDevice(MyTimerMsg);
    TimerOpen := FALSE
  END;

  IF MyTimerMsg # NIL THEN
    DeleteExtIO(MyTimerMsg,SIZE(TimeRequest));
    MyTimerMsg := NIL
  END;

  IF MyMsgPort # NIL THEN
    DeletePort(MyMsgPort);
    MyMsgPort := NIL
  END;
END EndPause;

(*-----------------------------------------------------------*)

BEGIN

  MyMsgPort  := NIL;
  MyTimerMsg := NIL;
  TimerOpen  := FALSE;

END Pause.
```

DEFINITION MODULE Notify

```
DEFINITION MODULE Notify;

PROCEDURE Oops( text : ARRAY OF CHAR);

END Notify.
```

IMPLEMENTATION MODULE Notify

```
IMPLEMENTATION MODULE Notify;

FROM SYSTEM     IMPORT BYTE, ADR;
FROM Intuition  IMPORT IntuiText, AutoRequest, IDCMPFlagSet;
FROM Graphics   IMPORT JAM1;

PROCEDURE Oops( text : ARRAY OF CHAR);

VAR
  msg,
  cont : IntuiText;
BEGIN
  WITH msg DO
    FrontPen  := BYTE(0);
    DrawMode  := JAM1;
    LeftEdge  := 10;
    TopEdge   := 15;
    IText     := ADR(text);
    ITextFont := NIL;
    NextText  := NIL;
  END;
  WITH cont DO
    FrontPen  := BYTE(0);
    DrawMode  := JAM1;
    LeftEdge  := 5;
    TopEdge   := 3;
    IText     := ADR("Continue");
    ITextFont := NIL;
    NextText  := NIL;
  END;
  IF AutoRequest(NIL, ADR(msg), NIL, ADR(cont),
                 IDCMPFlagSet{},
                 IDCMPFlagSet{}, 320, 60) THEN END;
END Oops;

END Notify.
```

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 warranties 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 this leading Amiga program author to continue his outstanding work.

AC's TECH AMIGA   Amazing Computing AMIGA   AC's GUIDE AMIGA

Name______________ ____________________________________

Address __________ ____________________________________

City__________________________ _____ State _____ ZIP__________

Charge my □Visa □MC #____________________ ______________

Expiration Date ______________ Signature __________ ______________

MasterCard or VISA

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 many 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, ***Amazing Computing*** at a savings of over $23.00 off the newsstand price!	□ □ □	$24.00 US $44.00 Foreign Surface $34.00 Canada and Mexico
One Year of AC SuperSub!	**Save over 46%** 12 monthly issues of ***Amazing Computing*** PLUS AC's GUIDE AMIGA 2 Product Guides a year! A savings of $31.80 off the newsstand price!	□ □ □	$36.00 US $64.00 Foreign Surface $54.00 Canada and Mexico
Two Years Of Amazing	**Save over 59%** 24 monthly issues of the number one resource to the Commodore Amiga, ***Amazing Computing*** at a savings of over $56.80 off the newsstand price!	□	$38.00 US (sorry no foreign orders available at this frequency)
Two Years of AC SuperSub!	**Save over 56%** 24 monthly issues of ***Amazing Computing*** PLUS AC's GUIDE AMIGA 4 Complete Product Guides! A savings of $75.60 off the newsstand price!	□	$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

Back Issue Volumes: Volume 1-$19.95* Volume 2-$29.95* Volume 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. Airmail rates available.

***Introducing:* AC's TECH AMIGA January 1991 Issue: $14.95**

One-Year Subscription to *AC's TECH* —Four Issues!

Charter Rate: $39.95 (limited time only)!

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 V3.8 & V3.9
AC#3...Source & Listings V4.5 & V4.6
AC#5...Source & Listings V4.9
AC#7...Source & Listings V4.12 & V5.1
AC#8...Source & Listings V5.4 & V5.5

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#9...Source & Listings V5.2 & 5.3
AC#10...Source & Listings V5.6 & 5.7
AC#11...Source & Listings V5.8, 5.9 & 5.10
AC#12...Source & Listings V5.11, 5.12 & 6.1

InNOCKulation Disk: Virus protection

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 410; 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.

Silent Binary Rhapsodies

Robert Tiess

00000000

A switch is activated << CLICK >>;
    it glows a cool, steady redness.
Disk drives whirl, then programs process
    Two megabytes of free RAM wait,
slowly dwindling down to mere K's
    Lists, lines appear on CRT
While a programmer waits to see
    Laboured creations execute;
while this happens, microchips mule
    whip and shift data bits around,
compu-magic chants without sound...

00000001

Silicon cities small, thriving
    with millions of quick bits surging,
grouping in masses, form bytes great,
    traveling at impulse's rate
through circuits closed and open gates,
    mini messengers with one fate:
serving CPU's in their state,
    pushing, popping stacks, regulate
seas of flowing data, create
    static registers, relocate
and assemble machine code mates,
    to disassemble and delete.

00000010

Pixel by pixel a screen fills
    with colorful characters' frill
upon a glossy blackbackround;
    Messages appear; alarms sound;
a software insect has been found!
    Programs stumble, die, crash abound.
Analysis time: Bugs confound,
    compile and thwart users, astound
even clever "hackers," wrapped and wound
    in jarring jargon— hunters bound
quick to quest as bug-thirsty hounds.

00000011

Running across a wooden desk
    is a white mouse, doing its task
of pointing and clicking, running
    around on smooth pad fast, guiding
arrow about a screen, swinging
    its tail with vital haste, tasting
options from a menu, pulling-
    down, picking-up ICONs, calling
files while gadgets start prompting
    press a key... a sound chip sings
a MIDI symphony. Growing
    chaos succeeds multitasking.

00000100

Hungry printers swallow paper
    as important output tapers
to serial or parallel scores
    of bytes black on white scrolls, that pour
liquid reams blank from reservoirs.
    Tractors turn, friction-feeds firm soar
and spew pages of text, while more
    ribbon fades to gray. An encore
performance commences with roars
    of dot matrix spurts, and four
daisy wheels spin crisp print before
    laser-wielding writers conquer.

00000101

These binary brandishing beasts
    of technology have at least
afforded mankind a light year
    in an inch, pinching cosmos clear
to the size of an atom mere,
magnifying microns to peer
    at as if miles. Computers shear
the universe into frontiers
    logical and bring the far near,
shortening distance, saving dear
    time, making burdens dissappear.

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 □ Canada/Mexico $52.95 □ Foreign Surface $56.95 □
1 year of AC	12 big issues of Amazing Computing! Save over 45% off the cover price!	US $24.00 □ Canada/Mexico $34.00 □ Foreign Surface $44.00 □
1-year SuperSub	AC + AC's GUIDE – 14 issues total! Save more than $31 off the cover prices!	US $36.00 □ Canada/Mexico $54.00 □ Foreign Surface $64.00 □
2 years of AC	24 big issues! Save over 55% off the cover price! US only.	US $38.00 □
2-year SuperSub	28 big issues! Save more than $75! US only.	US $69.00 □

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

AC's TECH AMIGA Reader Service Card

Name ___
Street ___
City ___ State ___ ZIP ___
Country ___

AC's TECH – Premiere Issue
Valid until 3/15/91
See page 81 for reference numbers

A. At your option, please provide the following data:
○ 1. Male ○ 3. Married
○ 2. Female ○ 4. Single/never married
○ 5. Separated/Widowed

B. What is your age?
○ 6. Under 18 ○ 9. 35-44
○ 7. 18-24 ○ 10. 45-64
○ 8. 25-34 ○ 11. 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 2500 016. Amiga 500
014. Amiga 3000 017. do not own an Amiga

D. Where do you use your Amiga(s), and how many hours per week do you use an Amiga at each location?
018. At home ___ hours per week
019. At work ___ hours per week
type of business ___
020. [illegible] ___ hours per week
type of business ___
021. At school ___ hours per week
class taken ___
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 ___

H. How much money are you likely to spend on all Amiga products and peripherals this year?
[illegible]
029. Less than $250 034. $2501-$5000
030. $250-$500 035. $5001-$7500
031. $501-$1000 036. $7501-$10000
032. $1001-$1500 037. Over $10000
033. $1501-$2500

I. How did you obtain this issue of AC's TECH?
038. Preordered this issue prior to publication.
039. Ordered this issue after seeing announcement.
040. Ordered a charter subscription.
041. Purchased it at my Amiga dealer.

J. How many other individuals including yourself have read or will read this issue of AC's TECH?
042. ___ others in addition to myself

K. Overall, how would you rate this issue of AC's TECH in the following categories?
Editorial content: 043. Excellent 044. Good 045. Fair 046. Poor
Disk: 047. Excellent 048. Good 049. Fair 050. Poor

L. 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.

M. Have you ever purchased a copy of AC's GUIDE?
054. Yes 055. No

301

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 – Premiere Issue
Valid until 3/15/91
See page 81 for reference numbers

A. At your option, please provide the following data:
○ 1. Male ○ 3. Married
○ 2. Female ○ 4. Single/never married
○ 5. Separated/Widowed

B. What is your age?
○ 6. Under 18 ○ 9. 35-44
○ 7. 18-24 ○ 10. 45-64
○ 8. 25-34 ○ 11. 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 2500 016. Amiga 500
014. Amiga 3000 017. do not own an Amiga

D. Where do you use your Amiga(s), and how many hours per week do you use an Amiga at each location?
018. At home ___ hours per week
019. At work ___ hours per week
type of business ___
020. At school ___ hours per week
type of business ___
021. At school ___ hours per week
application ___
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 ___

H. How much money are you likely to spend on all Amiga products and peripherals this year?
[illegible]
029. Less than $250 034. $2501-$5000
030. $250-$500 035. $5001-$7500
031. $501-$1000 036. $7501-$10000
032. $1001-$1500 037. Over $10000
033. $1501-$2500

I. How did you obtain this issue of AC's TECH?
038. Preordered this issue prior to publication.
039. Ordered this issue after seeing announcement.
040. Ordered a charter subscription.
041. Purchased it at my Amiga dealer.

J. How many other individuals including yourself have read or will read this issue of AC's TECH?
042. ___ others in addition to myself

K. Overall, how would you rate this issue of AC's TECH in the following categories?
Editorial content: 043. Excellent 044. Good 045. Fair 046. Poor
Disk: 047. Excellent 048. Good 049. Fair 050. Poor

L. 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.

H. Have you ever purchased a copy of AC's GUIDE?
054. Yes 055. No

302

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's TECH/AMIGA
P.i.M. Publications, Inc.
P.O. Box 869
Fall River, MA 02722-9969

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's TECH/AMIGA
P.i.M. Publications, Inc.
P.O. Box 869
Fall River, MA 02722-9969

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.

3 THREE NEW PRODUCTS FROM ICD

Flicker Free Video™

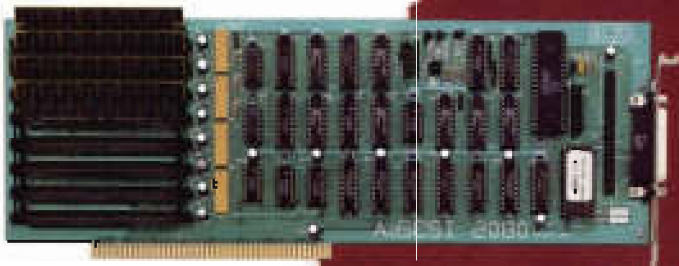
With *Flicker Free Video* (FFV) and a standard VGA or multi-frequency monitor, any Amiga® 500, 1000, or 2000 computer can produce a high quality display, free of interlace flicker and visible scan lines. Installation requires no soldering or advanced technical knowledge and frees the video slot in Amiga 2000 computers for other uses. *FFV* is compatible with all software, works in low and high resolutions interlaced or not, and has no genlock conflicts. *FFV* uses a multi–layer circuit board and surface–mounted components, packing a lot of power into a very small space. Both PAL and NTSC are automatically recognized and fully supported. Full overscan is supported, not just a limited overscan. Three megabits of random access memory are used to ensure compatibility with overscan screens as large as the Amiga can produce.

Ad*Speed*™

ICD expands its line of innovative enhancement products for the Amiga with the introduction of Ad*Speed*, a full featured 14.3 MHz 68000 accelerator for *all* 68000–based Amiga computers. Ad*Speed* differs from other accelerators by using an intelligent 16K static RAM cache to allow zero wait state execution of many operations at twice the regular speed. *All* programs will show improvement. Ad*Speed* will make your Amiga run faster than any 68000 or 68030 accelerator without on-board RAM. Ad*Speed* works with all 68000 based Amiga computers, including the 500, 1000, and 2000. Installation is simple and requires no soldering. Ad*Speed* has a software selectable true 7.16 MHz 68000 mode for 100% compatibility — your computer will run as if the stock CPU was installed. 32K of high speed static RAM is used for 16K of data/instruction cache and 16K of cache tag memory. A full read and write-through cache provides maximum speed.

AdSCSI™ 2080

The fastest, most versatile SCSI host adapter (hard drive interface) available for the Amiga 2000 now comes in a new configuration. *AdSCSI 2080 is not* DMA, but its clean design and advanced caching driver provide greater throughput than any available DMA interface. All the features you want are included at no additional charge: autoboot from Fast File System partitions, Commodore® SCSIDirect and Rigid Disk Block conformance for no mountlist editing and compatibility with third party SCSI devices, and the most advanced removable media support available, including automatic DiskChange and no partitioning restrictions. *AdSCSI 2080* also includes sockets for adding two, four, six, or eight megabytes of RAM using 1 megabyte SIMMs. If expansion slots are in high demand, then this card could be your answer.



Flicker Free Video, *AdSpeed*, and *AdSCSI 2080* join ICD's existing and growing line of power peripherals and enhancements for Amiga computers. Our experience and expertise allow us to give you the products and support you deserve. From beginning to end, every possible aspect of product development and production is handled in-house. We design all the hardware, lay out all the circuit boards, and write all the software. We assemble and test our products in our own facility, providing us with an unmatched level of control over the finished product. It is never out of our hands. These are more examples of the advantages you get from ICD. The best product. At the best price. With the best support. No compromises.

ICD

1220 Rock Street
Rockford, IL 61101
(815) 968-2228 Information
(800) 373-7700 Orders only
(815) 968-6888 FAX

Flicker Free Video, FFV, AdSpeed and AdSCSI are trademarks of ICD, Inc. Commodore is a registered trademark of Commodore Electronics Limited. Amiga is a registered trademark of Commodore-Amiga, Inc.

Circle 122 on Reader Service card.

Nothing is faster!

Amiga Loads Faster

Shown here: ALF RC1 RLL

Filerunner Card (ST506 or SCSI)
An internal Hard Disk System with a pass-through adaptor for use with the Amiga 2000. This system comes already built and installed with a 3.5" hard disk drive. Each card can accommodate a second drive (SCSI up to seven) for future expansion, and the power supply is strong enough to run two drives simultaneously. Formatted (low level/ high level) with Interleave 1:1. Data Transfer Rate: 420 KB/sec. Just plug it in and switch it on.

Filerunner Box (flat version)
An external Hard Disk System for use with the Amiga 500 or 1000. This Hard Disk System comes already built and installed in a flat case which fits neatly under the monitor. Each case can accommodate a second drive for future expansion, and the power supply is strong enough to run two drives simultaneously. Just connect it and switch it on.

All systems include:
- **Framebackup** which allows the boot block to be copied onto a separate floppy disk allowing easy start up after a crash.
- **AddFlp** – our newest feature which installs up to 10 floppy partitions onto the harddisk allowing diskcopy between the disk and the floppy.

Capacity
- 30MB up to 180MB
- up to 4 GigaBytes on special order.

For more information see your dealer or contact us.

Pre'spect Technics Inc.
P.O. Box 670, Station 'H'
Montreal, Quebec H3G 2M6
Phone: (514) 954-1483
Fax: (514) 876 2869

NEW! Non-flicker Cable
No need to invest in an expensive monitor for your Amiga! This handy, inexpensive cable illiminates 90% of annoying flicker on any TTL monitor.