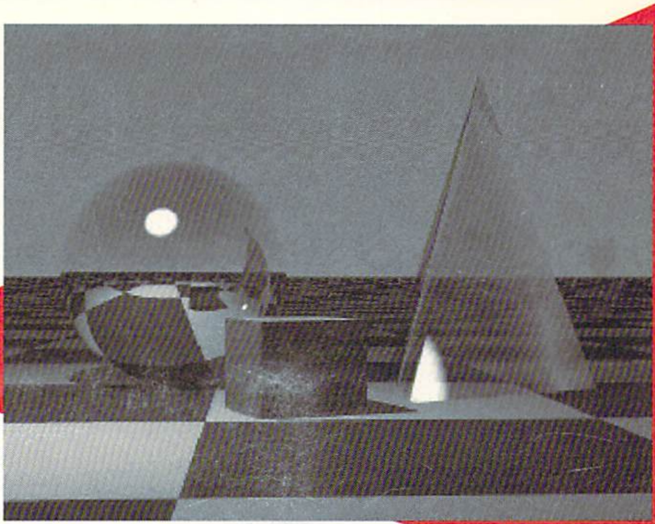
Disk Inside

Disk Inside

source and executable code for all articles

AMIGA WORLD TECH JOURNAL


Write your own ray tracer; see page 25.

On Disk

(See page 33)

2.0 include files: ***straight from the CATS' mouth***

Post: ***a PostScript interpreter***

Plus source code and executables for articles

U.S.A. $15.95
Canada $19.95

Volume 1, Number 3 August/September 1991 An IDG Communications Publication

Captain's Log 217:2055
Time shift equalization routines recoded in Aztec C. All systems go. It's good to be back home.
Space Staton 2055 - Viido Polikarpus

New Release
Aztec C 5.2

- **Amiga**
- **Macintosh**
- **MS-DOS**
- **80x86, 680x0 ROM**

Call Today for Special Discount Prices and Technical Information

Aztec C is available for Amiga, Macintosh, MS-DOS, Apple II, TRS-80, and CPM/80. Native and cross development.

Aztec Embedded C hosts on MS-DOS and Macintosh. Targets include: 68000 family, 8086 family, 8080/Z80/64180, & 6502

Aztec C includes - C compiler, assembler, linker, librarian, editor, source debugger, and other development tools.

MANX 800-221-0440

• Outside USA: 908-542-2121 FAX: 908-542-8386 • Visa, MC, Am Ex, C.O.D, domestic & international wire • one and two day delivery available

Manx Software Systems • 160 Avenue of the Commons • Box 55 • Shrewsbury NJ 07702

Linda Barrett Laflamme, *Editor*
Louis R. Wallace, *Technical Advisor*
Barbara Gefvert, Beth Jala, Peg LePage, *Copy Editors*

Peer Review Board
Rhett Anderson, John Foust, Eugene Mortimore
Gary Bonham, Jim Goodnow, Bryce Nesbitt
Gene Brawn, Scott Hood, Carolyn Scheppner
Brad Carvey, David Joiner, Leo Schwab
Joanne Dow, Sheldon Leemon, Tony Scott
Keith Doyle, Dale Luck, Mike Sinz
Andy Finkle, R.J. Mical, Richard Stockton
Steve Tibbett, John Toebes, Dan Weiss

Mare-Anne Jarvela, *Manager, Disk Projects*

Laura Johnson, *Designer*
Alana Korda, *Production Supervisor*
Debra A. Davies, *Typographer*

Kenneth Blakeman, *National Advertising Sales Manager*
Barbara Hoy, *Sales Representative*
Michael McGoldrick, *Sales Representative*
Giorgio Saluti, *Associate Publisher, West Coast Sales*
2421 Broadway, Suite 200, Redwood City, CA 94063
415/363-5230
Heather Guinard, *Sales Representative, Partial Pages*
800/441-4403, 603/924-0100
Meredith Bickford, *Advertising Coordinator*

Wendie Haines Marro, *Marketing Director*
Laura Livingston, *Marketing Coordinator*
Margot L. Swanson, *Customer Service Representative, Advertising Assistant*
Lisa LaFleur, *Business and Operations Manager*
Mary McCole, *Video Sales Representative*

Susan M. Hanshaw, *Circulation Director, 800/365-1364*
Pam Wilder, *Circulation Manager*

Lynn Lagasse, *Manufacturing Manager*

Roger Murphy, *President*
Bonnie Welsh-Carroll, *Director of Corporate Circulation & Planning*
Linda Ruth, *Single Copy Sales Director*
Debbie Walsh, *Newsstand Promotion Manager*
William M. Boyer, *Director of Credit Sales & Collections*

Stephen C. Robbins, *Vice-President/Group Publisher*
Douglas Barney, *Editorial Director*

The AmigaWorld Tech Journal (ISSN 1054-4631) is an indepedent journal not connected with Commodore Business Machines, Inc. *The AmigaWorld Tech Journal* is published bi-monthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603-924-0100. Entire contents copyright 1991 by IDG Communications/Peterborough, Inc. No part of this publication may be printed or otherwise reproduced without written permission from the publisher. **Postmaster**: Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

MESSAGE PORT

We're all teachers.

I'M PLEASED THAT so many of you want to share your expertise with fellow readers. Since we started our conference on BIX (aw.techjournal, stop by), one of the most common questions I see is "How can I write for the *Tech Journal*?"

The best way to start is to have a concrete idea and to draw up a rough outline of the article. Mail the outline along to me either by traditional means or via BIX (llaflamme) and include the article's projected length, a list of accompanying diagrams or listings, and a writing sample. Be sure to mention your credentials and background in the cover letter. If we like the idea, I'll call and request the complete article and code.

Remember, the *Tech Journal*'s charter is to provide articles and listings that teach the reader how to better use the Amiga's unique capabilities. The stress is on Amiga; an introductory language piece will never make it beyond the "Thank you, but no thank you" bin. Past issues of the *Tech Journal*, CATS' *AmigaMail*, and DevCon notes are good sources of inspiration.

Don't get so busy writing, however, that you can't take time out for the 1991 North American Developers Conference. Slated for September 4–7 at Colorado's Denver Marriott City Center, this year's conference should have a full-to-bursting schedule of technical seminars, considering all the new hardware, 2.0, and CDTV developments. As they were last year, marketing and public relations sessions are also on the agenda. *Registered* developers should receive registration forms and information automatically. If you haven't, contact CATS, 1200 Wilson Dr., West Chester, PA 19380.

While CATS is educating the developers, the Amiga Developers' Association is rallying to educate the rest of the computing world about the Amiga's strengths. Under the leadership of a new board, the ADA is focusing on a single goal: to promote the Amiga to all that will listen. The first target is the computer press. As ammunition, the board will use position papers on the Amiga's benefits for users and developers. (Any potential authors out there?) The ADA also wants to pass along profiles of companies and individuals using the Amiga for innovative applications and unique situations. (If you are, promote yourself. Contact the ADA.) To offer your help, contact Chairperson Al Hospers at Dr. T's Music Software, 220 Boylston St. #306, Boston, MA 02167. If you'd like more general information or a membership application (dues are $150 per year), write to Treasurer Jerry Wolosenko, ADA Membership Drive, Psygnosis, 29 St. Mary's Court, Brookline, MA 02146.

Whether educating programmers and engineers about the latest techniques and technologies or educating the world at large about the Amiga's power, we're all working for the same goal: to advance and expand the the Amiga market. Get in the game and help. ■

[illegible]

On Disk

ARexx Arcana: Hosts and Quotes

By Marvin Weinstein

EVERYONE AGREES AREXX (which is based on IBM's REXX) lets you harness the power of multitasking by relaying instructions between programs. A cloud of mystery and confusion, however, hovers over the concept of an ARexx host program. The number of ARexx-capable applications that can serve as ARexx hosts is large and growing rapidly. As a result, understanding how ARexx processes a command and passes it to a host is vital. The key to this is understanding ARexx's syntax and the way in which ARexx handles quotes.

SYNTAX OVERVIEW

Because a REXX command is defined in terms of what it is not, a lightning review of REXX syntax is in order. The simplest meaningful statement that can appear in a REXX program is a clause. The ARexx interpreter recognizes six types of clauses:

A Comment

A statement enclosed between a /* and its matching */, which is usually ignored by REXX. Every REXX program, however, must begin with a comment. (On IBM mainframes the comment was used to distinguish REXX programs from scripts written in IBM's EXEC language.)

An Instruction

A statement that begins with one of 32 keywords. The 32 instructions, combined with assignment statements, define the skeleton of all REXX programs. They determine the manner and order in which things happen. For example, the SAY instruction causes ARexx to type a line of text to the CLI that launched the ARexx exec (program).

```
/* beginning comment */
say hello
```

produces:

```
HELLO
```

A Label

A clause that consists of a single name followed by a colon, such as:

```
START:
```

You use labels to indicate the beginning of subroutines or, in special circumstances, to modify the flow of control in a REXX program.

An Assignment

Any clause of the form:

```
name = expression
```

where a variable name is followed by an = and an expression to be evaluated.

A Null

A blank line or a line containing only a semicolon.

A Command

Any clause that does not fall into one of the preceding categories. When ARexx encounters a command clause it evaluates the expression and then passes the result to an external host for further action. The external host may be either the native operating system or an application program designed to receive and act upon ARexx messages. You can modify the default host for an ARexx program using the ADDRESS instruction. A large portion of problems come from misunderstanding who the default host for a program is, and what limitations that host operates under. (More on this later.)

The fact that REXX is typeless makes it easy to learn. No need to specify variables in advance; as far as REXX is concerned all variables are strings and defined, whether or not they have appeared on the left side of an assignment statement. If a variable has not been assigned a value, by default REXX assigns it the string obtained by uppercasing the variable's name. For example, it is REXX's job to figure out if the variable r in the statement:

```
x = 4*r
```

has a value for which the * operation is well defined. If r has not appeared in an assignment statement, REXX uppercases the r to obtain:

```
X = 4*R
```

and recognizes that it does not know how to multiply a number by a letter. The program then fails and produces an error message.

Because the REXX interpreter treats all strings that do not contain blanks as possible variables, it automatically uppercases text as it goes along. To tell ARexx not to examine a string for possible substitutions, we enclose it in matching single or double quotes. Thus, the program:

Avoid misquoting commands and the host trouble that follows

by learning some of ARexx's more esoteric facets.

```
/**rexx:sayit.rexx
*
**/
say Hello world
say "Hello world"
say 'Hello world'
say "Hello" world
```

produces the output:

```
HELLO WORLD
Hello world
Hello world
Hello WORLD
```

Because you can use either single and double quotes, it is easy to include a quote inside a text string. For example:

```
/** rexx:quotetest.rexx
*
**/
say "This is an example of a string with a quote ' "
say 'This is an example of a string with a quote " '
```

produces the output:

```
This is an example of a string with  quote '
This is an example of a string with a quote "
```

Another mechanism for including a quote in a string is to type either '' or "". In this case REXX eats the first quote and prints the second. For example:

```
/** rexx:escapetest.rexx
*
**/
say 'This is an example of a string with a quote " '
say "This is an example of a string with a quote "$kern4pt"  "
```

produces the same output as the quotetest.rexx example.

This simple discussion illustrates why quoting is used and the need for adopting a flexible set of quoting conventions. As long as you limit yourself to manipulating strings within ARexx programs, there is nothing more to know. Peculiarities arise, however, when you send strings containing quotes to an ARexx host. You must understand how ARexx processes a command line and the quoting conventions required by the host application to avoid problems caused when host applications use the same quoting characters as ARexx.

COMMANDS AND HOSTS

To guarantee that ARexx will immediately recognize a line as a command clause, you enclose it in quotes. In fact, the IBM REXX manual recommends this as good programming style. As an alternative, you could begin the line with a pair of matched quotes. For example, if an ARexx exec contains the following:

```
""dir
"dir"
dir
```

ARexx instantly recognizes the first two lines as commands, but treats the third differently. First ARexx checks to see if dir is a built-in instruction. Because this test fails, ARexx next looks in the rexx: directory for a file named dir.rexx or dir. Not finding one, it passes dir to the default host to see if the host knows what to do with the command.

Who is the default host for the program and will it know how to field this command? Things are starting to get complicated. The default host for an ARexx program depends upon how the ARexx program was launched. Even for programs launched from a shell, there is a difference between those programs launched using rx and programs launched without using rx, from a truly ARexx-integrated shell.

As ARexx arrived after the original Amiga CLI was developed, it is not as well integrated into the CLI environment as REXX is on an IBM mainframe running VM/CMS. One trivial, but annoying, aspect of this lack of integration is that under AmigaDOS 1.3 you cannot launch an ARexx program by typing only its name. To run an ARexx program called myprog.rexx from a CLI (or Shell) you must type:

```
rx myprog
```

Under AmigaDOS 2.0 the situation is better, but not great. If you set the script bit for an ARexx program and add the file's directory to your path, you can execute the program by typing only the full name of the file, including the .rexx extension. This is a minor annoyance.

A more serious complication is that under both AmigaDOS 1.3 and AmigaDOS 2.0 the script is actually launched using rx. The only difference is that in one case the rx is explicit. While being able to omit the rx command is cosmetically nicer, in both cases the default host for the process is not the underlying CLI (or Shell) but REXX. As proof, run the following program:

```
/** rexx:addresstest.rexx
*
**/
say "Hi, my address is"
say address()
```

►

using rx addresstest. Because REXX is the default host, you encounter some unexpected problems. For example, what happens if you run the program:

```
/**rexx:mydir.rexx
*
**/
"dir"
```

by typing:

```
rx mydir
```

The correct answer is nothing. In interpreting the program, ARexx recognizes the single command clause and passes it to the default host, which is REXX itself. The second time REXX gets the command without quotes, because it ate one level of quotes when it interpreted the string. REXX now attempts to resolve dir as an instruction, and so on. To avoid an infinite loop, it will no longer pass the command to itself, therefore failing to resolve the command. The result is that nothing happens.

The situation is quite different if you run the same program from a truly ARexx integrated shell, such as the WShell replacement shell marketed by William Hawes, the creator of ARexx. WShell knows about ARexx. When you type something such as mycommand, WShell automatically searches the rexx: directory for an ARexx program with the name mycommand.rexx or mycommand. This makes launching ARexx programs as simple as launching ordinary executables. Second, if you run the program addresstest.rexx from a WShell, the output will be something like:

```
WSH_1
```

which is different from what happens in an Amiga Shell. This occurs because each WShell has an ARexx port and can serve as the default host for an ARexx program launched from it. If you run the program mydir.rexx from a WShell, ARexx recognizes "dir" as a command clause, strips off the quotes and passes dir to the default host, WSH_1, which treats dir as if you entered it from the keyboard. This makes issuing system commands much more transparent.

"A simple mechanism for telling an ARexx exec to execute a string as a system command is the ADDRESS instruction."

A simple mechanism for telling an ARexx exec to execute a string as a system command is the ADDRESS instruction. Use it to change the default host for dir to be AmigaDOS. You may place the ADDRESS COMMAND instruction either in front of the command clause to be executed by AmigaDOS or on a separate line. When ADDRESS COMMAND appears on the same line as the command clause, it changes the default host to COMMAND for only that clause; when the instruction appears on a separate line, then it changes the default host to COMMAND for all subsequent command clauses. Both methods are illustrated below. Try running each example from a CLI (or Shell) to see what happens. If you have WShell, launch each of them with and without using rx. No matter how these programs are launched, the command dir >ram:dirlist executes correctly; only the results returned by the ADDRESS function differ.

```
/** rexx:dirtofile1.rexx
*
**/
say address()
address command 'dir >ram:dirlist'
say address()
```

or, equivalently:

```
/** rexx:dirtofile2.rexx
*
**/
say address()
address command
'dir >ram:dirlist'
say address()
```

(Note that the command clause must be in quotes in these examples or ARexx will attempt to interpret the line and break on the : and the >.)

While it may appear that using the ADDRESS COMMAND instruction to run system commands is only slightly more cumbersome than having a WShell as a default host, appearances are deceiving. Things get more complex when you wish to launch a series of commands that must be executed sequentially.

SUBTLETIES WITH MULTIPLE COMMANDS

Consider the problem of printing a file with an ARexx exec using a program that requires a stack size of 20000 when your default stack is 4000. If you run it from a WShell, the following program will work:

```
/** rexx:texprint.rexx
*
**/
arg filename
"stack 20000"
"dvilj -o par: work:dvifiles/"filename".dvi"
exit 0
```

In this case the command clauses are sequentially issued to the same host, say WSH_1. If, however, you modify the program to:

```
/** rexx:texprint2.rexx
*
**/
arg filename
address command
"stack 20000"
"dvilj -o par: work:dvifiles/"filename".dvi"
exit 0
```

it will break because the stack will be too small for dvilj. The COMMAND keyword in the ADDRESS instruction tells ARexx to execute each command as an independent process. Thus, effectively, the command:

```
stack 20000
```

is executed in one Amiga shell, and the command:

```
dvilj -o par: work:dvifiles/"filename".dvi
```

is issued in another. This would appear to be a serious limitation to running ARexx without WShell, but fortunately there is a simple solution. Create a single string of commands, concatenating individual commands separated by the line-feed character "0A"x. As in:

```
string = "command 1"||"0a"x||command 2||"0a"x||. . .
```

Now, send this one string to a shell using:

```
address command string
```

This trick is very useful for ARexx execs that cannot depend upon the existence of WShell.

A more interesting problem is to open a CLI to show the output of the dviljp command as it runs, and then close the shell when finished. In this case, the previous approach needs a somewhat arcane modification, as shown below for OS 2.0:

```
/** rexx:printit.rexx
*
**/
arg filename
string = "stack 20000 +"||'0a'x
string = string||"run dvilj -o par: "filename".dvi +"
string = string||'0a'x||"endcli"
runstring = "run > con:10/10/100/100/printing.../close "||string

address command runstring
exit 0
```

A + is needed after each command because the AmigaDOS run >con:... construction redirects the output of the command series to a CLI; thus, now the multiple string command belongs to run and AmigaDOS 2.0 rules apply. Without the + characters, the stack 2000 command would go to the newly opened CLI and everything else would go to the shell that launched the program.

"The names of all ARexx ports should be all uppercase."

ADDRESS CHANGES

ARexx-capable applications have one or more public message ports to which you can send ARexx. When ARexx programs are launched from within an application, the default host is customarily set to be one of these ports. If the program needs to interact with more than one application, then you must repeatedly change the default host. To do so, you use one of the ADDRESS instruction's four general forms:

```
ADDRESS {string|symbol}   expression
ADDRESS {string|symbol}
ADDRESS VALUE  expression
ADDRESS
```

You are already familiar with a special case of the first two forms from the previous examples that used ADDRESS COMMAND, either by itself or on the same line as the command clause. While ADDRESS can take either COMMAND or VALUE as a keyword, all other strings that immediately follow the ADDRESS instruction are interpreted as the name of a public message port. Thus:

```
address TURBOTEXT7
address 'TxEd Plus1
address 'ProVector'
address FASTHOST
address AmigaTeX
```

redirect subsequent commands to the indicated message ports.

Be sure to note that ARexx does not interpret the string that immediately follows the ADDRESS instruction. If the string is enclosed in quotes, ARexx uses it as is; otherwise ARexx automatically converts it to uppercase. Because ARexx conducts a case-sensitive search for public message ports, you must provide quotes whenever needed or the search will fail. ARexx does interpret the expression that follows the port name.

A common mistake with the ADDRESS instruction is to try something clever, such as:

```
/* program-specific code here */
myprogram = "ProVector"
/* more code here */
address myprogram  command
```

This trick fails to accomplish the desired result because ARexx attempts to pass the command to a port called MYPROGRAM, instead of ProVector. The VALUE keyword tells ARexx to interpret the symbol or string that follows, before constructing a port name. Hence:

```
address value myprogram
```

causes all subsequent commands to be forwarded to the port ProVector. Given the way in which the ARexx parser works, however, the instruction:

```
ADDRESS VALUE expression3
```

cannot be followed by a command clause on the same line. After ADDRESS VALUE is used, the default host for the program is changed. To make restoring the original default host simple, ARexx keeps both the new host and the previous host in memory. Using the ADDRESS instruction by itself toggles between these two values. Therefore:

```
address command
/* program-specific code here */
n = 7
myprogram = "TURBOTEXT"n
address value myprogram
/* more code here */
address
say address( )
```

produces the output

```
COMMAND
```

Actually, you can use a trick to accomplish the same thing as the VALUE keyword without changing the default host. Use the INTERPRET instruction. A construction such as:

```
interpret address myprogram command
```

will work just as well.

The Amiga style guidelines recommend that the names of ARexx ports should be all uppercase, which makes a lot of sense. First, it avoids the need for quoting the string that follows the ADDRESS instruction, which is a common source of problems with ARexx programs. Second, when you write complicated programs that launch string execs containing an ADDRESS instruction, every extra required quote hurts. ►

(Note: The fact that the true name of a public message port is all uppercase does not prevent you from enhancing readability by typing names in mixed case. ARexx automatically converts the name to uppercase before sending any messages, unless forced not to.)

Unfortunately, not all current applications conform to the style guide's recommendations. For this reason you must know the correct name of a message port to send it a message. Its name should be in the application's manual. If not, you can run the program and then type:

```
rx "say showlist('p')"
```

to view a list of all public message ports.

FLEXIBLE QUOTING NEEDED

A final class of quoting problems arises when you try to include ARexx string programs inside command clauses. As an example, consider one of the earliest ARexx-capable Amiga applications, the editor TxEd Plus. TxEd Plus lets you add or delete menu items that are capable of launching ARexx execs. The syntax of the command used for this purpose is:

```
MENU n K "Menu Text"  ARexx_program
```

where n is a menu number from 0 to 5, K is a letter specifying the keyboard alternative for the menu item, the text that follows is what appears in the menu, and ARexx_program specifies the name of the program to be run. At first glance this is not a bad syntax, but problems arise because TxEd Plus requires that quotes be used for its quoting characters.

Suppose that you wish to run an ARexx exec to add new menu entries. In that case, to add an entry to menu 5, with keyboard alternative Z, which launches the exec CalcStuff/calc.rexx, you have to type:

"The notion of a string program is unique to ARexx."

```
MENU 5 Z '"Launch Calc"' "CalcStuff/calc.rexx"
```

Why do you need two sets of quotes enclosing Launch Calc? ARexx will eat the first set of quotes when it processes the line, and TxEd Plus requires the string that follows to be enclosed in double quotes if it contains spaces. Even the order of the quotes used in this example is forced upon you; if you type "'Launch Calc'" you will not obtain the desired outcome. The program name must also be enclosed in quotes so that ARexx does not attempt to do a division before passing the result to TxEd Plus. Because the quotes are eaten by ARexx, TxEd Plus receives a valid program name to pass to REXX whenever the new menu item is selected.

While this example is not too bad, consider what happens if you replace the name of an ARexx program with a string program. A string program is an ARexx invention that allows you to execute short ARexx execs directly by typing:

```
rx "line1 ; line2 ; ... "
```

where the semicolons are REXX's end-of-line character. Either single or double quotes are acceptable as delimiters for a string program. String programs let you avoid having a large number of small ARexx files associated with an exec that creates multiple menu entries. Because an enclosing set of quotes must survive after ARexx has eaten the outermost quotes in the example, you must use all of your tricks to achieve that end. The example below will work:

```
MENU 5 B '"Begin numbered points"' , "'INSERT ""\pointbegin"";
  ""txedstuff/newline""'"
```

While this sort of program looks considerably more mysterious than the preceding one, it is not difficult to understand. Remember that ARexx first processes the line before handing it on. This means that the outermost quotes in each string disappear and, simultaneously, the double-double quotes are converted to single-double quotes before the line is passed to TxEd Plus. When TxEd Plus receives it the line looks like:

```
MENU 5 B "Begin numbered points"'INSERT "\pointbegin";"txedstuff/
  newline"'
```

Examine the file startup.txed for more examples.

This sort of programming would be much simpler if a more flexible way of handling the quoting problem had been adopted by the host application. One way to increase flexibility would be for the host to allow additional quoting characters, as well as a way to handle program strings that contain more than one line, as indicated by the presence of semicolons. An example of a program that handles this sort of scripting in a more general manner is provided by the terminal program VLT. Another way to overcome this kind of difficulty is to include keywords for the host to parse on instead of using quotes. An example of this sort of approach is taken in the editor program called TurboText. Obviously, many strategies can be adopted to handle the proliferation of quotes that appear when people try to stuff string programs into command clauses; the common feature of all solutions is that the host application has to be more flexible in its quoting than ARexx.

FINAL ADVICE

ARexx has abilities that transcend those possessed by REXX on any other platform. With these powers, however, comes added complexity. Understanding how command clauses get passed to a host application is imperative. The ADDRESS instruction includes a mechanism for sending ARexx messages to public message ports, but you must remember that port names are case sensitive and that you need the VALUE keyword to use a computed string in the ADDRESS instruction. Finally, the notion of a string program is unique to ARexx. While this is a very powerful facility, exploiting it can lead to bizarrely complicated quoting patterns. The key to working out these complexities is to keep in mind:

- ARexx eats one level of quoting each time it processes a line.
- ARexx converts escaped quotes (double-double or double-single quotes) to ordinary quotes each time it processes a line.
- You must also know how the host application handles the processed line when it receives it.

If in a given project you find it difficult to sort out quoting complexity, then making liberal use of the tracing console and ARexx's built-in tracing commands can provide you with valuable insights into what is happening. ■

Marvin Weinstein uses ARexx and REXX extensively in his work at the Stanford Linear Accelerator. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (mweinstein).*

This is the most cost effective way to increase the speed of your computer. AdSpeed™!

Ad*Speed*
ICD expands its line of innovative enhancement products for the Amiga® with the introduction of **Ad*Speed***, a low cost, full featured 14.3 megahertz 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. No 68000 or 68020 accelerator without on board RAM will make an Amiga run faster.

Ad*Speed* continues ICD's tradition of providing the best product available. These are some of the features that set it apart from the rest:

- Works with all 68000–based Amiga computers, including the 500, 1000, and 2000.
- Simple no solder installation — just remove the computer's 68000 and plug **Ad*Speed*** into its socket.
- Low power, high speed CMOS 68000 CPU for full 100% instruction set compatibility.
- Software selectable speeds, with a true 7.16 megahertz mode for 100% compatibility. Switches speeds on the fly without rebooting the computer.
- 32 kilobytes of high speed static RAM — 16K of data/instruction cache and 16K of cache tag memory.
- Full read and write–through cache for greatest speed.
- Bus monitoring to prevent DMA conflicts.
- ICD's famous quality, dependability, and support.
- Worlds smallest 68000 accelerator. (Photo above is actual size).

ICD, Incorporated 1220 Rock Street Rockford, IL 61101 USA
(815) 968-2228 Information (800) 373-7700 Orders (815) 968-6888 FAX

On Disk

Floating-Point Math Alternatives

By John Foust

CHOOSING THE WRONG floating-point math model for your C program could cut performance in half, double or triple memory consumption, or introduce numeric error in calculations. It might even prevent your program from running on some Amigas. Balancing these trade-offs and picking the best model requires an understanding of AmigaDOS's floating-point math libraries, how a C compiler translates floating-point operations to assembly language, and how a compiler's link libraries perform math operations.

First, take a look at what your choices are. Several of the system libraries contain functions that perform math operations with IEEE (Institute of Electrical and Electronic Engineers) and FFP (Fast floating-point) floating-point numbers. Table 1 lists the system math libraries and the type of functions they contain. The base math libraries perform simple operations (addition, multiplication, negation, and so on), while the transcendental libraries contain functions such as sine and cosine, plus functions to translate numbers between IEEE and FFP formats. (For a complete list of math library functions see the *Amiga ROM Kernel Reference Manual: Includes & Autodocs*, Addison-Wesley.)

The Motorola fast floating-point system stores floating-point numbers in 32 bits. The IEEE 754 standard includes both single-precision values in 32 bits and double-precision values in 64 bits. Motorola's 68881 and 68882 numeric coprocessor chips use the IEEE format. Table 2 shows each representation's minimum and maximum values for numbers (using C's version of scientific notation), the digits of precision (relevant number of digits after the decimal point), along with the storage size.

The Motorola coprocessors and the Amiga OS double-precision IEEE libraries use double-extended numbers for their intermediate calculations. These numbers use 80 bits of storage. Both SAS and Manx C store these types in 96 bits (12 bytes) of data storage. They could be stored in just ten bytes, but storing them in three 32-bit longs is faster and more convenient, especially on the 68020.

OLD STYLE OR STANDARD

What's the difference between FFP and IEEE? FFP math is quite fast, even though the calculations are done in software. It does not offer double precision, however, or speed up if a coprocessor is available. The IEEE format offers both single and double precision for more freedom and control over floating-point calculation and storage. If your programs are going to exchange binary data with other types of computers, IEEE is a good choice. Although, IEEE is about twice as slow as FFP, it will perform as fast or faster than FFP if a coprocessor is present.

Your C compiler adds another level of complexity to your choice. In Standard C programs, there are three possible floating-point data types: float, double, and long double. Standard C allows a compiler to represent the float, double, and long double types in the same way. (It recommends a floating-point representation with a range similar to single-precision IEEE, meaning that the FFP model is not quite good enough for Standard C.) Having both floats and doubles represented by single-precision values is perfectly legal under the FFP models and is supported in Manx and SAS C. Even under IEEE math models, SAS C has compiler switches to set the representation of float and double types. With these, you can represent all floating-point types as either single- or double-precision, or split-single and double types as different precisions.

In an expression that combines floating-point variables with built-in operators such as +, -, *, and /, the compiler evaluates the expression in a predictable way, to generate assembly language and data to represent the floating-point values. In evaluating the expression, a conversion between types can take place: integers can be converted to floats, floats to doubles, and so on, before the rest of the expression is evaluated. Promotion from one type to another is defined to progress from int to float to double to long double. For example, in an expression involving an int and a double (and no other type casts), the integers are promoted to doubles, and the entire expression is performed in double precision.

When calling a function with floating-point arguments, similar promotion can take place. In old-style C, all floating-point arithmetic was defined to take place in double precision. In a function call, all float parameters were promoted to double, then pushed on the stack. Under Standard C, a function's prototype determines if floats will be promoted to double before being pushed on the stack. In the absence of a function prototype, promotion takes place as in old-style C. If a Standard C function prototype says the argument is a float, it stays a float.

Take a look at the theory in action. Listing 1 shows a very short C function that loads a constant into a double, multiplies that double by two, then takes the cosine of it and stores it into a long double. In math.h, the cos() function is declared as accepting a double and returning a double. Listing 2 shows the assembly language generated for this function, as compiled for FFP math and generated by the Manx 5.0 compiler. Listing 3 shows the IEEE version. Listing 4 shows it under double-extended IEEE, and Listing 5 shows it with direct in-line 68881 codes.

Examine these listings side by side and you will quickly see

Table 1

Amiga library	Contents
mathffp.library	FFP math functions
mathtrans.library	FFP transcendental functions
mathieeesingbas.library	IEEE math operator library, uses coprocessor if present
mathieeesingtrans.library	IEEE transcendental library, uses coprocessor if present
mathieeedoubbas.library	IEEE double precision library, uses coprocessor if present
mathieeedoubtrans.library	IEEE double transcendentals, uses coprocessor if present

Table 2

Type	Minimum	Maximum	Digits of Precision	Size in bits
FFP	5.4e-20	9.2e19	6-7	32
Single IEEE	1.3e-38	3.4e38	6-7	32
Double IEEE	2.2e-308	1.8e307	15-16	64
Double Extended IEEE	3.4e-4932	1.2e4932	19	80 (96)

the difference in the code generated under each math model. In the FFP listing, no distinction is made between float, double, and long double. Simple 32-bit values are loaded into registers, and such internal subroutines as jsr .fmul# are called to perform math. In the IEEE and double-IEEE list

Listing 1: Original function()

```
#include <math.h>
function()
{
float a;
double b;
long double c;
    b = 3.14159;
    a = b * 2;
        c = cos(a);
}
```

Listing 2: FFP

```
; #include <math.h>
;
; function()
; {
xdef    _function
_function:
link    a5,#.2
movem.l .3,-(sp)
; float a;
; double b;
; long double c;
;
;
; b = 3.14159;
    move.l  #$c90fd042,-12(a5)
;
; a = b * 2;
    move.l  -12(a5),d0
    move.l  #$80000042,d1
    jsr .Fmul#
    move.l  d0,-4(a5)
;
; c = cos(a);
    clr.l   -(sp)
    move.l  -4(a5),-(sp)
    jsr _cos
    add.w   #8,sp
    move.l  d0,-20(a5)
;}
.4
    movem.l (sp)+,.3
    unlk    a5
    rts
.2  equ -20
.3  reg
;
;
    xref    _cos
    xref    .begin
    dseg
    end
```

Listing 3: IEEE

```
; #include <math.h>
;
; function()
; {
    xdef    _function
_function:
    link    a5,#.2
    movem.l .3,-(sp)
; float a;
; double b;
; long double c;
;
;
; b = 3.14159;
move.l  #$f01b866f,-8(a5)
move.l  #$400921f9,-12(a5)
;
; a = b * 2;
move.l  -8(a5),d1
move.l  -12(a5),d0
move.l  #$00000000,d3
move.l  #$40000000,d2
jsr .Pmul#
jsr .dtof#
move.l  d0,-4(a5)
;
; c = cos(a);
move.l  -4(a5),d0
jsr .ftod#
move.l  d1,-(sp)
move.l  d0,-(sp)
jsr _cos
add.w   #8,sp
move.l  d1,-16(a5)
move.l  d0,-20(a5)
;}
.4
movem.l (sp)+,.3
unlk    a5
rts
.2  equ -20
.3  reg d2/d3
;
;
xref    _cos
xref    .begin
dseg
end
```

►

Listing 4: Double-Extended IEEE

```
; #include <math.h>
;
; function()
; {
xdef  _function
_function:
link  a5,#.2
movem.l .3,-(sp)
; float a;
; double b;
; long double c;
;
;
; b = 3.14159;
move.l  #$f01b866f,-8(a5)
move.l  #$400921f9,-12(a5)
;
; a = b * 2;
move.l  -8(a5),d1
move.l  -12(a5),d0
move.l  #$00000000,d3
move.l  #$40000000,d2
jsr .Pmul#
jsr .dtof#
move.l  d0,-4(a5)
;
; c = cos(a);
move.l  -4(a5),d0
jsr .ftod#
move.l  d1,-(sp)
move.l  d0,-(sp)
jsr _cos
add.w  #8,sp
jsr .dtox#
move.l  .p0+8,-16(a5)
move.l  .p0+4,-20(a5)
move.l  .p0,-24(a5)
;}
.4
movem.l (sp)+,.3
unlk  a5
rts
.2 equ -24
.3 reg d2/d3
xref  .p0
;
;
xref  _cos
xref  .begin
dseg
end
```

Listing 5: Direct 68881

```
; #include <math.h>
;
; function()
; {
xdef  _function
_function:
link  a5,#.2
movem.l .3,-(sp)
fmovem.x  .4,-(sp)
; float a;
; double b;
; long double c;
;
;
; b = 3.14159;
move.l  #$f01b866f,-8(a5)
move.l  #$400921f9,-12(a5)
;
; a = b * 2;
fmove.d -12(a5),fp0
fmul.d #"$4000000000000000",fp0
fmove.s fp0,-4(a5)
;
; c = cos(a);
fmove.s -4(a5),fp0
fcos.x  fp0
fmove.x fp0,-24(a5)
;}
.5
fmovem.x  (sp)+,.4
movem.l (sp)+,.3
unlk  a5
rts
.2 equ -24
.3 reg
.4 freg
;
;
   xref  .begin
   dseg
   end
```

ings, the double constants are twice as large as the FFP constants. Storing doubles in memory or on disk consumes twice the memory or disk space as using single precision. In the c= cos(a) line of the double-IEEE listing, note that it takes three long-word move.l instructions to store the long-double quantity. The direct 68881 listing is very different than the others. In it the compiler generates special 68881 coprocessor instructions (such as fmul.d and fcos.x fp0) instead of calling internal subroutines to perform the math. The limitation is you need a 68020 or 68030 CPU, plus a coprocessor chip to run the program.

COPROCESSOR COOPERATION

The 68020 and 68030 support direct, tight-coupled access to the numeric coprocessor chip. The 68000 supports only a slower interface, treating the 68881 coprocessor as a peripheral device. At one time Microbotics and CMI provided a replacement mathieeedoubbas.library that called their peripheral 68881s, allowing 68000-based Amigas to use the 68881 as a peripheral. Although these devices could double the speed of math operations, they never became commonplace. Unfortunately, few commercial applications used the IEEE double-precision library. As the Amiga market matured, math-intensive programs were usually offered in two forms: an integer- or FFP-based version that worked quickly on any Amiga and a version compiled for the Amigas equipped with a 68020 and 68882. Since Amiga OS 1.3, however, the double-precision libraries can take advantage of numeric coprocessor chips, as long as they declare a math resource during the autoconfig process. This means both 68000/68881 and 68020/68882 combinations give faster results, if present.

In Amiga OS 2.0, Commodore added a new mathieeesingbas.library that has the same functions as the mathffp.library, except that it uses a math resource if present. This library gives the speed of single-precision IEEE with the added benefit of using a coprocessor, if present. Under Amiga OS 2.0, the system will not auto-init the library, so it is not on the library list until an application asks for it. Under very low memory the library init could fail, therefore do not assume success. Check the result of the OpenLibrary() call before continuing.

LIBRARY LINGO

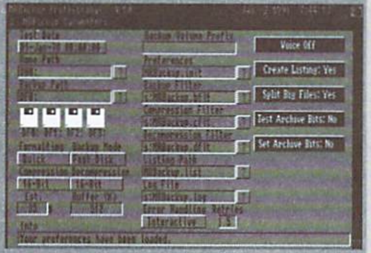
After the compiler turns the assembly language intermediate file to an object file, the linker comes into play. Do not be confused by the word "library." It has two contexts in a math models discussion: A given compiler has link libraries, and the Amiga OS has math libraries. An OS library is essentially a program that contains useful subroutines that any other program can call—the mathieeedoubbas.library in the LIBS: directory is an example. A compiler's link libraries are composed of object .o modules that contain definitions of ►

RXTools

RXTools is an object oriented interface builder which extends the capabilities of ARexx and the Amiga. With the built in editor, RXTools provides a complete development system for ARexx on the Amiga.

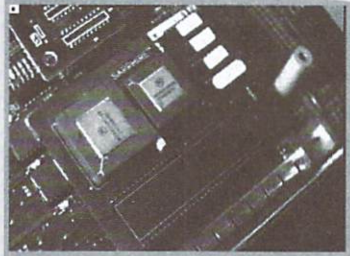
- ARexx function host environment system which runs in the background - ARexx Required.
- Allows you to create windows, gadgets, requesters and more within your ARexx applications.
- Provides a programming addition to ARexx not otherwise available.
- Built in editor allows easy manipulation of RXTools and ARexx scripts.
- AmigaDOS V1.3 and 2.0 Compatible.
- Retail Price $54.95

MRBackup Professional



Your preferences have been loaded.

MRBackup Professional allows for easy yet powerful backup commands at the click of a gadget. MRBackup Professional will allow backups to floppy, another hard drive, virtual file, SCSI streaming tape, or even to DAT tape.
• SCSI Streaming Tape Capability - Use your existing controller and a SCSI tape drive, NO Extra hardware required (most controllers and SCSI drives supported). • Utilizes full compression, (12 to 16 bit), to save backup space. - With FastDisk option, even floppy backups take less time. • Has full ARexx integration. - Allows for complete external control of the backup options.• AmigaDOS V1.3 V2.0 Compatible.
• Retail Price $54.95, With Tanberg 150Mb SCSI Streaming Tape External $949.95 Internal $749.95,
Wangtek 150Mb SCSI Streaming Tape External $999.95 Internal 799.95,
Sony full height DAT External $2049.95 Internal $1849.95.



Sapphire Accelerator

68020 / 68881

Fits snugly in an Amiga 1000, 500, and 2000
For easy installation, included is a disk with instructions, pictures and some public domain benchmark software. Also included is an electrostatic discharge (ESD) information card, and an ESD safety strap. Features include:

- Factory Installed 12 MHz 68020 CPU And 68881 FPU 32 Bit Processors operating at 7.16 MHz.
- Speed Increases Of Up To 2.4 Times Faster In Integer, And 3.2 Times Faster In Floating Point.
- Small Compact Size, Smallest Yet At Only 3 1/8" x 4 1/4" x 1/2".
- Not A Pseudo Accelerator, but a true 32 Bit Accelerator Card Using 32 Bit Processors.
- One Full Year Factory Warranty.
- Retail Price 399.95

TTR Development, Inc
6701 Seybold Rd. Suite 220
Madison, WI 53719

Sales And Technical Support
608-277-8071

BBS Support 608-277-8072

TR DEVELOPMENT INC.

INNOVATIVE SOFTWARE FOR THE COMPUTER INSIDE US ALL

Bix support under TTR.Support

Dealer Pricing Available!

such C library functions as fopen(). Other link libraries—one for each math model—contain subroutines that perform arithmetic operations on floating-point numbers. When you link your program's object modules, the final executable is a mixture of the functions that you wrote plus functions from the standard link libraries. If your program performs floating-point math, you must link with the proper library to perform those math operations.

To complicate things, a compiler's math link library can invoke functions in an OS library. An example is SAS C's lcmieee.lib link library. It opens the Amiga's IEEE library to perform math. (Also, note that both Manx and SAS C offer IEEE link libraries that do not open the system's libraries, but instead link in a stand-alone IEEE math implementation.)

By linking with the proper math link library, internal subroutines such as the jsr .fmul# are resolved. Listing 6 shows the source code for the internal subroutine .Fmul. It simply calls the SPMul() function in the Amiga's FFP library, inside the amiga_ffp# internal subroutine, which first checks to make sure that library is open and ready.

Listing 6: Direct 68881

```
; Copyright (C) 1986 by Manx Software Systems, Inc.
; :ts=8

    public .Fmul ]
.Fmul:
    move.l #_LVOSPMul#,-(sp)
    jmp amiga_ffp#
```

One common mistake is to list the link libraries in the wrong order when telling the linker how to produce an executable. The symptom is that floating-point numbers do not print from the printf() function. For example, in Manx C's IEEE model, you need to specify the ma.lib before the c.lib. Each math link library contains a variant of the printf() function that understands the math model. Be careful, the Standard C link library has a version of printf() that does not understand the %f format specifier. If the linker sees the c.lib printf() before the math library printf(), it links the wrong function into your executable.

THE CHOICE IS YOURS

This might seem obvious, but if your program does not perform any floating-point operations, then you do not need to link it with a math library. What defines a math operation? Any part of an expression involving a floating-point type, or a cast to a floating-point type, invokes internal math functions. For example, even though a program has no floating-point variables, an expression in the program might cast integer variables to floats to perform an accurate division by a fractional number.

As you can see in the assembly language listings, it is possible to initialize floating-point variables without linking with a math library, as long as you do not use any Standard C operators with these variables. If you open a math library on your own and call the SPMul() function yourself, there is no overhead of jumping through the compiler's stubs and internal subroutines.

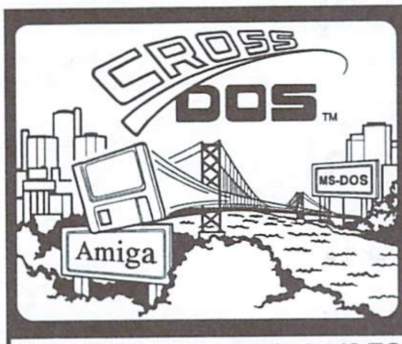
If you want raw speed, consider integer math. If your application can avoid floating-point and perform fixed-point integer arithmetic, chances are it will be faster than any floating-point implementation.

If you do need to use a library, the FFP libraries are about twice as fast as 1.3 IEEE libraries and give a reasonable amount of precision for most applications. OS 2.0's mathieeesingbas.library, however, complicates your choice. One of the main advantages of the FFP library was that it was in ROM and thus available on every Amiga. Because the library was also relatively fast for nonhardware-assisted math, it was the correct choice. With the arrival of high-end applicatons that need more accuracy and speed coupled with industry pressure for a standard floating-point format, CATS staffers recommend the mathieeesingbas.library which is built in ROM, almost as fast as the mathffp.library on non-coprocessor machines, and much faster on coprocessor-equipped Amigas.

If you prefer accuracy, standardization, and the added benefit of running much faster on enhanced machines, choose the IEEE libraries or consider direct 68881 instructions for the fastest possible math on enhanced Amigas. Also, there is no reason not to turn on your compiler's 68020 code generation, because all 68881-equipped Amigas have an advanced processor.

As with all program design choices, the "best" math model involves decisions about speed and storage. Keep your preferences and the above tips in mind, and the trade-offs between math models will be more apparent. ■

John Foust is president of Syndesis Corp. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*



CrossDOS™

...MS-DOS®

DISK ACCESS DONE RIGHT!

Version 4

Suggested Retail:

$39.95 (US)

The first MS-DOS File System for the *AMIGA*®

- Reads or writes any 360K or 720K MS-DOS or ATARI ST® disks (Version 2.0 or higher) with standard Amiga-connected floppy drives.
- Transparently accesses MS-DOS files from any utility or application (including file requesters) because it fully integrates into the Amiga operating system (AmigaDOS Versions 1.2, 1.3 and *2.0*).
- Provides an easy installation program.
- Format, diskcopy and check the integrity of an MS-DOS disk.

Break the 720K Barrier . . .

- Access many megabytes of data using standard removable cartridges and external hard drives. *(MakeAB and JLink files NOT supported)*
- Using your Amiga hard drive controller, you can access most MS-DOS formatted hard drive media just like MS-DOS floppies.

Evening and weekend BBS available for update information and to trade uploads or PD file conversion programs.

BBS# (313) 459-7271 -- 6 PM to 10 AM EST on weekdays and all day on weekends.

CONSULTRON

11280 Parkview
Plymouth, MI 48170

Technical Support
(313) 459-7271

Reviewed in many popular Amiga Magazines:

★★★★★ Rating
"CrossDOS is truly excellent. It's a wonder that no one thought of this wonderful idea ages ago."
David W. Martin
.info Magazine

"... CrossDOS is unquestionably the best utility I have seen for reading from and writing to 3.5-inch MS-DOS disks on the Amiga."
Tim Walsh
AmigaWorld

"... now I send files back and forth to and from IBM land in a totally transparent fashion."
Jim Locker
Amazing Computing

Amiga is a registered trademark of Commodore-Amiga, Inc. MS-DOS is a registered trademark of Microsoft, Inc. Atari ST is a registered trademark of Atari, Corp.

ASDG

MEANS COLOR!

AT ASDG...
OUR ONLY BUSINESS IS COLOR.

If you've produced a quality color imaging product...
You can choose no better publisher than ASDG.

CALL US AT (608) 273 - 6585

The GadTools Library

By Paul Miller

WORKBENCH 2.0'S PROFESSIONAL look and operational consistency are attributable primarily to the new gadget constructs available in the GadTools library. An integral part of this latest operating system, GadTools combines the look of 2.0 and aspects of several standard controls to form a set of new gadget classes. These gadgets can make your life much easier, and that's why GadTools is not just a tool—it is an interface programmer's friend!

The best thing about GadTools is its ease of use. GadTools lets you create an entire list of complicated gadgets with much less effort and hassle than you are used to. With one function and a few parameters, for example, you can create an entire scrollable list gadget, complete with slider and arrow buttons—GadTools automatically combines the necessary controls into a single gadget. No longer must you create a proportional gadget and combine it with two Boolean gadgets to make a scroller—simply ask for a Scroller GadTools gadget, and the system takes care of the busy work and maintenance for you!

GadTools is also extremely extensible. This means that as new gadget classes and parameters become available, you can easily add them using 2.0's tag facility. Tags are a means of providing information to many new 2.0 functions without filling out such large temporary structures as NewScreen and NewWindow. They also allow the easy addition of new features. If you omit tags, the system typically defaults to a group of standard settings. For that reason, you need set only the specific parameters you require.

Tags are specified either as an array of TagItems or as variable parameters—containing the tag and its data—to the function. Here is the format of the TagItem structure:

```
typedef ULONG Tag;

struct TagItem {
Tag ti_Tag; /* tag ID */
ULONG ti_Data;
};
```

TagItem lists are terminated with a tag of type TAG_DONE or TAG_END.

TAG—YOU'RE A GADGET!

In version 36 of gadtools.library, there are 13 kinds of gadgets. Each type has a set of tags that specify required or special information about the gadget.

Both the GENERIC_KIND (a standard gadget type) and BUTTON_KIND (an action button) accept the disable tag. This also applies to most other gadget types:

GA_DISABLED (BOOL; default = FALSE)
Set to TRUE to disable the gadget, FALSE otherwise.

The CHECKBOX_KIND is a state-selecting gadget. A checkmark appears when the gadget is activated. You can use the check tag with it:

GTCB_Checked (BOOL; default = FALSE)
The initial state of the checkbox (TRUE = CHECKED).

The CYCLE_KIND is a single button with multiple states. This gadget cycles forward through a list of strings each time you click on it. It also cycles backward through the list when you hold down the Shift key and click. Its corresponding tags are:

GTCY_Labels (STRPTR *)
A required tag, this is a NULL-terminated array of string pointers. These strings comprise the gadget's offerings.

GTCY_Active (UWORD; default = 0)
The ordinal number (counting from zero) of the initially active choice in the cycle.

INTEGER_KIND is a standard string-editing gadget that allows you to edit only integers. The Integer gadget accepts these tags:

GTIN_Number (ULONG; default = 0)
Determines the initial contents of the Integer gadget.

GTIN_MaxChars (UWORD; default = 10)
The maximum number of digits that the Integer gadget can hold.

A powerful gadget, LISTVIEW_KIND displays a list of items that you can select and scroll through. It has an optional scroll bar with arrows and an optional editing facility for the selected item. ListView accepts several tags:

GTLV_Top (UWORD; default = 0)
Determines the top item (ordinal) visible in the list.

GTLV_Labels (struct List *)
A required tag, this acts as a pointer to an Exec List of labels whose ln_Name fields are displayed in the list.

GTLV_ReadOnly (BOOL; default = FALSE)
If TRUE, ListView is a read-only gadget.

GTLV_ScrollWidth (UWORD; default = 16)
Determines the width of the scroll bar.

GTLV_ShowSelected (struct gadget *)
Set to NULL to have the currently selected item displayed beneath the list. Alternately, set to a pointer to an existing GadTools STRING_KIND gadget if you want an editable display of the currently selected item.

"GadTools lets you create an entire list of complicated gadgets with much less effort and hassle than you are used to."

GTLV_Selected (UWORD; default = ~0)
Ordinal number of the currently selected item, or ~0 for no current selection.

LAYOUTA_SPACING (default = 0)
Extra space to place between lines of the list.

The MX_KIND gadget provides a set of items that, like radio controls, you can select only one at a time. Its tags are:

GTMX_Labels (STRPTR *)
This tag is required. It serves as a NULL-terminated array of string pointers. These strings will appear as the labels beside the choices in the gadget.

GTMX_Active (UWORD; default = 0)
The ordinal number (counting from zero) of the initially active choice in the gadget.

GTMX_Spacing (UWORD; default = 1)
The amount of space between each choice of a set of mutually exclusive gadgets. This amount, added to the font height, produces the vertical shift between choices.

NUMBER_KIND is a read-only display gadget for integers. To it you can add:

GTNM_Number (default = 0)
A signed long integer to be displayed as a read-only number.

GTNM_Border (BOOL; default = FALSE)
If TRUE, this flag causes a recessed border to appear around the gadget.

The PALETTE_KIND gadget contains a display of all the available colors for the current screen, in addition to a box containing the "current" color—that is, the color selected from the gadget. The Palette gadget tags include:

GTPA_Depth (UWORD; default = 1)
Sets the number of bitplanes in the palette.

GTPA_Color (UBYTE; default = 1)
The palette color selected initially.

GTPA_ColorOffset (UBYTE; default = 0)
Determines which color to use first in the palette.

GTPA_IndicatorWidth (UWORD)
Determines the width of the current-color indicator. Set this if you want the indicator to appear above the palette.

GTPA_IndicatorHeight (UWORD)
Determines the height of the current-color indicator. Use this tag if you want to place the indicator to the left of the palette.

The SCROLLER_KIND gadget combines a proportional slider and two optional arrow buttons. The result is a ready-to-go scroller gadget that automatically tracks the position and size of the slider control. You have your choice of several tags for the Scroller gadget:

GTSC_Top (WORD; default = 0)
Determines what appears at the top of the area that the scroller represents.

GTSC_Total (WORD; default = 0)
The total number of items in the scroller area.

GTSC_Visible (WORD; default = 2)
The number of things visible in the scroller.

GTSC_Arrows (UWORD)
Attaches arrows to the scroller. The value supplied serves as the width of each arrow button for a horizontal scroller or the height of each button for a vertical scroller (the other dimension matches the whole scroller).

PGA_FREEDOM (default = LORIENT_HORIZ)
Determines whether the scroller is horizontal or vertical. Choose from LORIENT_VERT or LORIENT_HORIZ.

GA_IMMEDIATE (BOOL; default = FALSE)
Set to TRUE if you want to be notified of every GADGETDOWN event from the scroller.

GA_RELVERIFY (BOOL; default = FALSE)
Set to TRUE if you want to be notified of every GADGETUP event from the scroller.

The SLIDER_KIND combines a proportional gadget and, optionally, a numeric display that depicts the level or intensity of some value. Corresponding tags are:

GTSL_Min (WORD; default = 0)
Sets the minimum level of the slider.

GTSL_Max (WORD; default = 15)
Maximum level of the slider.

GTSL_Level (WORD; default = 0)
Current level of the slider.

GTSL_MaxLevelLen (UWORD)
Maximum length in characters of the level string when rendered beside the slider.

GTSL_LevelFormat (STRPTR)
A C-style formatting string for the slider level. Be sure to use the l (long) modifier. This string is processed using the EXEC RawDoFmt() function.

GTSL_LevelPlace (default = PLACETEXT_LEFT)
Indicates where the level indicator appears relative to the slider. Settings are PLACETEXT_LEFT, PLACETEXT_RIGHT, PLACETEXT_ABOVE, or PLACETEXT_BELOW.

►

GTSL_DispFunc ([LONG(*function)(struct gadget *, WORD)]; default = 0)
: This function calculates the level to be displayed. For example, a slider to select the number of colors might require a (1 << n) function here. Your function must take a pointer to a gadget as the first parameter, the level (a WORD) as the second, and return the result as a LONG.

GA_IMMEDIATE (BOOL; default = FALSE)
: Set to TRUE if you want to be notified of each slider GADGETDOWN event.

GA_RELVERIFY (BOOL; default = FALSE)
: Set to TRUE if you want to be notified of each slider GADGETUP event.

PGA_FREEDOM (default = LORIENT_HORIZ)
: Set to LORIENT_VERT or LORIENT_HORIZ to create a vertical or horizontal slider.

STRING_KIND is a standard alphanumeric string-editing gadget. The tags it supports are:

GTST_String (STRPTR; default = NULL)
: Specifies the initial contents of the string gadget. Set it to NULL if the beginning of the string is to be empty.

GTST_MaxChars (UWORD)
: The maximum number of characters that the string gadget is to hold.

TEXT_KIND, the last type of gadget, is a read-only alphanumeric text display. For tags you can choose among:

GTTX_Text (default = NULL)
: Serves as a pointer to a NULL terminated string to be displayed as a read-only text-display gadget or NULL.

GTTX_CopyText (BOOL; default = FALSE)
: This flag instructs the text-display gadget to copy the supplied text string instead of continuing to refer to the address of the supplied string.

GTTX_Border (BOOL; default = FALSE)
: If TRUE, this flag causes a recessed border to surround the gadget.

SET UP AND AWAY

Because GadTools resides in a ROM library, you must do some initial setting up before you can actually create a gadget. GadTools also requires a pointer to a VisualInfo block, which you can get by accessing either the Workbench screen or a custom screen. Finally, because Workbench 2.0 allows the user to choose almost any font as the standard, GadTools is very font-size specific and thus must know your font choice so it can place gadget text properly. You will typically want to force Topaz 80, so ask for it explicitly. See the setup() function in the demo program (in the Miller drawer of the accompanying disk) for the initialization described here.

Now you are ready to create a gadget list. Because all GadTools gadgets are dynamically allocated, GadTools provides a simple way for linking, adding, refreshing, and freeing entire lists of gadgets. To define your gadget list, declare a pointer to a gadget structure and call the GadTools CreateContext() function with the pointer's address:

```
struct gadget *glist = NULL;
struct gadget *gad;
gad = CreateContext(&glist);
```

This allocates a private gadget that describes the list. It also forms a base to which you can add more gadgets, and on which you can perform gadget-adding and -refreshing operations. Use the returned gadget pointer as a foundation for building your gadget list with CreateGadget().

CreateGadget() is the variable-arguments version of the gadget-allocation function and takes the gadget type, a pointer to the previous gadget, a pointer to a NewGadget structure, and the GadTools gadget tags as its parameters. The physical description of the gadget is specified in the NewGadget structure:

```
struct NewGadget
   {
   WORD ng_LeftEdge, ng_TopEdge; /* upper-left */
   WORD ng_Width, ng_Height; /* size */
   UBYTE *ng_GadgetText;  /* text label */
   struct TextAttr *ng_TextAttr; /* label font */
   UWORD ng_GadgetID;  /* gadget ID */
   ULONG ng_Flags;  /* special flags */
   APTR ng_VisualInfo;  /* VisualInfo */
   APTR ng_UserData;  /* your data */
};
```

Now, you must assign the VisualInfo pointer (obtained earlier) to the ng_VisualInfo field, and assign a declared TextAttr structure to the ng_TextAttr field. Then, finally, you are ready to create a gadget.

Fill out a NewGadget structure with the position and size of the gadget, a title, and an ID, along with the Font and VisualInfo information. To create a simple gadget of type BUTTON_KIND, you simply pass the required information:

```
gad = CreateGadget(BUTTON_KIND, gad, &newgad, TAG_DONE);
```

For a more complex type, say a Scroller gadget, you might want to add some more tags:

```
gad = CreateGadget(SCROLLER_KIND, gad, &newgad,
   PGA_FREEDOM, LORIENT_VERT,
   GTSC_Total, 20,
   GTSC_Visible, 10,
   GTSC_Arrows, 16,
   GA_RELVERIFY, TRUE,
   TAG_DONE);
```

To allocate a complete list of gadgets, begin with the gadget pointer returned from your CreateContext() call. Make the necessary adjustments to your NewGadget structure between calls to CreateGadget(), and then pass the gadget pointer returned from each call as the previous pointer in the following call. You need not check for allocation problems until the last gadget is requested, because CreateGadget() will simply return a NULL pointer if it receives NULL as the previous gadget pointer.

Once you have your gadget list, you can add it to a window and refresh the imagery with the standard Intuition gadget functions. Then you must call GT_RefreshWindow() on the window, so special GadTools imagery can receive the proper treatment. Additionally, new GT_BeginRefresh() and GT_EndRefresh() functions are provided to replace the standard Intuition window-refreshing functions.

FURTHER EVENTS

GadTools can also make dealing with gadget events much simpler. Using the new GT_GetIMsg() and GT_ReplyIMsg()

Continued on p. 61

Inside SCSI

Your system probably uses this interface every day, but do you know how it works?

By Greg Berlin

AS MORE COMPLEX tasks are required of microcomputers, system designers face the challenge of integrating all the functions required of sometimes dissimilar components, while maximizing system performance and minimizing the cost and effort involved in putting the system together. The current method of choice is the Small Computer Systems Interface (SCSI, pronouned scuzzy). The SCSI bus provides a standard means of connecting multiple intelligent peripherals to a host computer (see Figure 1). Thus, you can add many different types of devices (large hard drives, tape drives, CD-ROMs, and so on) to the host computer without modifying the generic system hardware or software.

While almost everyone has heard of SCSI, even many veteran computer users may not fully understand what SCSI is all about. First of all, the term small in SCSI's name implies that its application is confined to microcomputers and that its performance is limited. (The pronunciation "scuzzy" has not enhanced the standard's image, either.) As you will see, despite these misnomers, SCSI is indeed a powerful interface and well worth investigating thoroughly. It has found applications in machines spanning the entire spectrum of performance and price.

SCSI CONCEPTS

A very basic, yet important point to understand is that the SCSI bus is an I/O (input/output) bus, as opposed to a system bus or a device-level interface bus. A system bus (such as ZORRO II/III, NuBus, STD BUS, IBM PC BUS, or MULTIBUS I/II) is generally the place to add expansion cards. Device-level interface buses (including ST-506, ESDI, SMD, and QIC-36) provide the means to take physical control of a device, via analog or digital signals, to manage such tasks as head positioning and reading and writing raw data. Device-level interfaces can be attached directly to the system bus. A good example is the common ST-506/412 controller that plugs into the IBM PC BUS. Usually up to two ST506/412-compatible disk drives can be plugged into this controller board. While this method minimizes immediate cost, it limits its expandability and versatility. If you want to add a tape drive, for instance, you need to purchase another device-interface controller card to support the tape-drive interface ►

SCSI History

The SCSI interface's roots go back as far as the early 1960s. IBM's popular mainframes at that time made use of a byte-wide parallel I/O bus, known as the block multiplexer channel, that was capable of block transfers. Recognizing the need for an interface standard, the ANSI X3T9.3 committee began to define one in the early 1980s. Despite the popularity of IBM's block multiplexer channel, the ANSI committee decided not to accept it unchanged, due perhaps to the NIH (not invented here) syndrome or to political pressures from IBM's competitors. The standard that ANSI was developing was called the intelligent peripheral interface (IPI). The IPI bus was basically functionally equivalent to the block multiplexer channel, with a number of additions.

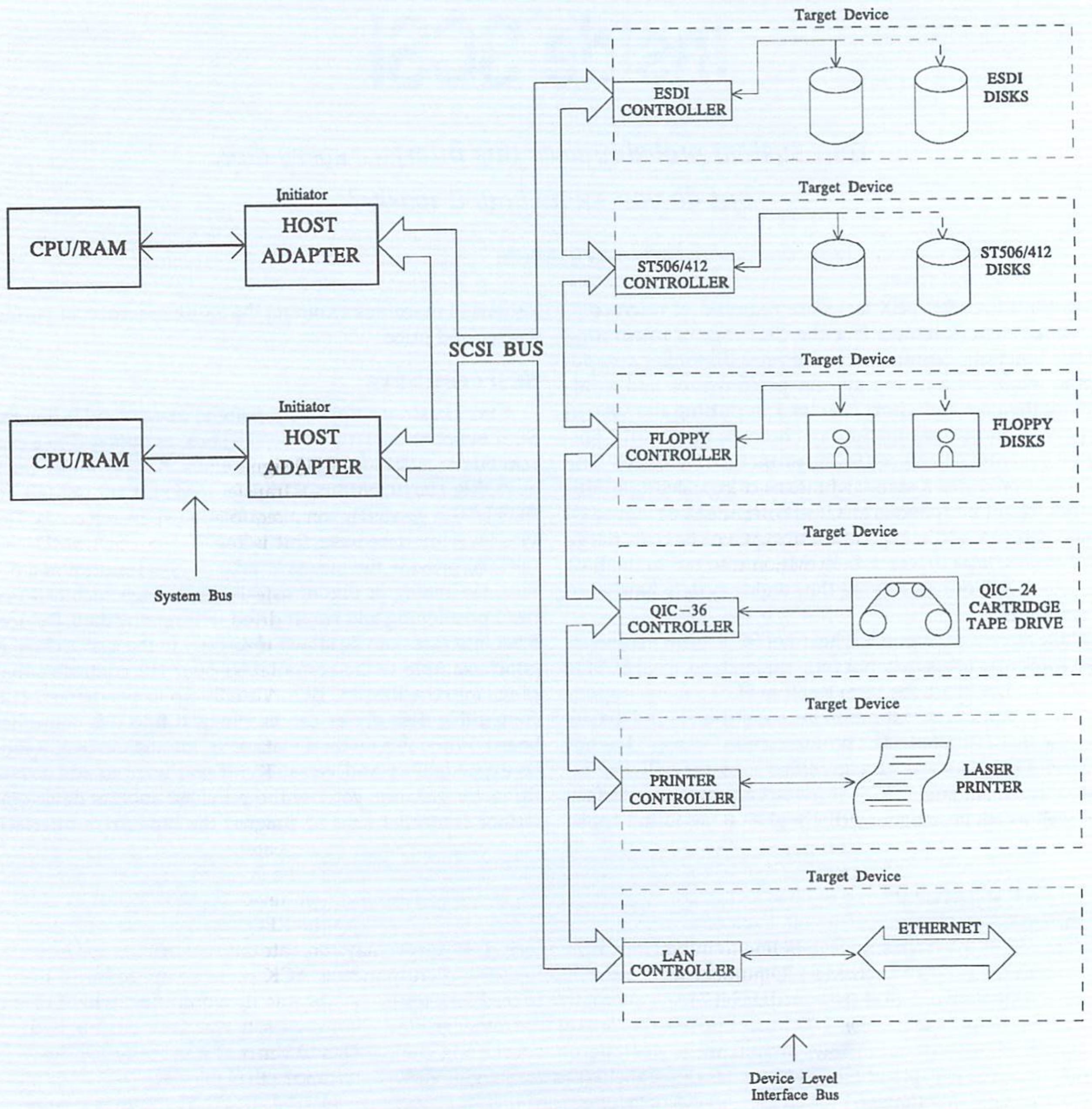
As an alternative to the block multiplexer channel of IBM, other groups were working on developing their own parallel I/O bus. Shugart Associates developed the Shugart Associates System Interface (SASI). Shugart was in the business of manufacturing disk drives, and several other drive manufacturers followed the company's lead and adapted the SASI interface to their drives. Because several of the key drive manufacturers adopted this interface, it became relatively widespread.

Shugart obviously had a great interest in seeing their parallel interface bus, instead of the IPI bus, adopted by the ANSI X3T9.3 committee. When it appeared that SASI would lose the battle, Shugart renamed its bus interface "SCSI" and submitted it to the ANSI X3T9.2 committee, where the competition was less entrenched. (The ANSI X3T9.2 committee deals with low-level interfaces.)

In 1984 the ANSI committee completed the SCSI-1 specification, and it was published in its final form in 1986. Further improvements and refinements in the standard have resulted in the recent release of the SCSI-2 specification, which is currently in its final draft review (and may be published by the time you read this). For a copy of the SCSI-2 draft proposal (document X3.131-198X), contact Global Engineering Documents, 2805 McGaw, Irvine, CA 92714, 800/854-7179.

—GB

Figure 1: A Multiple-Host and Multiple-Peripheral SCSI System



(such as QIC-36). Systems that require several disks or other peripherals can benefit from the added SCSI bus I/O layer. SCSI offers advantages in flexibility and system performance. You can connect several different peripherals to a single SCSI I/O bus, and the peripherals can communicate directly with each other. Bus speed is certainly not a limiting factor, as SCSI transfer rates have now reached up to 40 megabytes per second (MB/sec).

The SCSI bus can support up to eight devices connected to it. At first glance this may seem rather limiting, but the allowance in the spec for eight logical units per device and 256 logical subunits per logical unit provides more than enough expansion.

Each of the devices on the SCSI bus must be assigned a unique ID number, which is usually set with jumpers on the device. The SCSI ID serves two purposes: It uniquely identifies a device on the bus and it defines the device's priority during bus arbitration (the higher the device number the higher its priority).

Each of the eight possible devices is defined as an initiator, a target, or both. An initiator is part of the SCSI host adapter that connects a host computer system to the SCSI bus. In a typical system, a single initiator connects with one or more targets. A more complex system may have more than one SCSI host adapter (multiple initiators). This type of system can not only establish communication from any processor to any peripheral, but also from a host to host, because the host adapter is itself a SCSI device and can be a target as well as an initiator. Two peripherals (targets), however, cannot normally communicate with each other, and only two devices (one initiator and one target) can communicate with each other over the bus at any one time.

A host adapter consists of the hardware and software required to connect to the SCSI bus as an initiator, as well as the hardware and software to interface this logic to the host CPU. With the proliferation of single-chip SCSI controllers that can operate as both initiators and targets, constructing a SCSI host adapter has become quite easy. Typically, these single-chip devices attach directly to the SCSI bus on one side and, with a minimum of logic, are adaptable to any system bus on the other side. The SCSI-chip-to-system-bus interface can be as simple as software-polled I/O or more complex to enable high-speed data transfer via DMA. These single-chip controllers accept high-level commands and relieve the host CPU of the burden of manipulating and monitoring the SCSI bus signals.

Host system software is simplified because it no longer has to be concerned with the physical attributes of the device. The SCSI interface uses logical rather than physical addressing for all data blocks.

SCSI BUS PHASES

The SCSI protocol uses eight distinct logical phases: BUS FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA, STATUS, and MESSAGE. The last four are collectively known as the information transfer phases. The SCSI bus can be in only one of the eight states at a time.

The BUS FREE phase signifies that no SCSI device is actively using the SCSI bus, and that the bus is available for use. This phase is entered after a system reset or after the RST signal resets the bus. The BUS FREE phase is indicated by the deassertion of the BSY and SEL signals.

The ARBITRATION phase is entered when a SCSI device wishes to gain control of the bus as a master. This occurs when an initiator wants to select a target or a target wants to reselect its initiator. The ARBITRATION phase can be entered only from the BUS FREE phase. After a device determines that the bus is free, ARBITRATION begins by asserting the BSY signal and the SCSI ID of the device making the request on the appropriate data bit. Each of the eight possible SCSI devices uses a single data line to indicate its presence. The device with the highest ID number wins the arbitration and gains control of the bus (data bit 7 has the highest priority and bit 0 the lowest).

The SELECTION phase allows an initiator to select a target for the purpose of initiating a target function, such as a READ or WRITE command. In SCSI-2 the SELECTION phase is always entered following an ARBITRATION phase. SCSI-1 supported a single initiator option that eliminated the need for bus arbitration, so SELECTION could be entered following a BUS FREE phase. In both cases, the target is selected when the initiator puts the ID number of the target on the data lines and asserts the SEL signal.

The optional RESELECTION phase occurs when a target device wants to reconnect with the initiator that previously sent it a command. This phase takes place basically the same as the SELECTION phase, except the I/O line is asserted along with the SEL signal so that it can be distinguished.

The COMMAND, DATA, STATUS, and MESSAGE phases are grouped together as information-transfer phases because they are all used to transfer data or control information via the data bus. The C/D, I/O, and MSG signals are used to distinguish between the different information transfer phases (see Figure 2). The target drives these signals and therefore controls all changes from one phase to another. The REQ/ACK (and REQB/ACKB) lines are used to control the transfer of data between the target and initiator during the information-transfer phases.

The actual data transfer can be accomplished via synchronous or asynchronous protocols. Both methods make use of the ACK and REQ signal lines to perform the handshaking. The synchronous transfer mode is optional for a target device. An initiator can request that a target perform synchronous transfer, but if it is rejected then asynchronous mode is used.

When data is to be transferred to the initiator in asynchronous mode, the target drives the data lines and asserts the REQ signal. Data is held until the ACK signal is received from the initiator. The next data is then provided on the bus, and the process is repeated. When transferring data in the opposite direction, the target asserts the REQ signal, indicating that it is ready to receive data. The initiator drives the data lines and then asserts the ACK signal. The initiator continues to drive the same data until the REQ signal goes false. The target then negates the REQ signal, the initiator provides new data, and the process is repeated.

If the devices agree to the synchronous mode during the message phase, the target device does not wait for the ACK signal before sending the REQ signal for the next data. Instead, the target may generate one or more REQ pulses without the corresponding ACK pulse received (up to a pre-agreed maximum known as the REQ/ACK offset). Upon generating all the REQ pulses, the target compares the number of REQs and ACKs to verify that each data set was successfully received. To set up the synchronous data transfer, the devices establish a REQ/ACK offset (the maximum allowed number of REQs without corresponding ACKs received during the transfer) and the transfer period. The trans- ►

Figure 2: Information-Transfer Phases

MSG	C/D	I/O	Phase Name	Description
0	0	0	DATA OUT	Data sent from initiator to target
0	0	1	DATA IN	Data sent from target to initiator
0	1	0	COMMAND	Command sent from initiator to target
0	1	1	STATUS	Status sent from target to initiator
1	0	0	(reserved for future use)	
1	0	1	(reserved for future use)	
1	1	0	MESSAGE OUT	Message from initiator to target
1	1	1	MESSAGE IN	Message from target to initiator

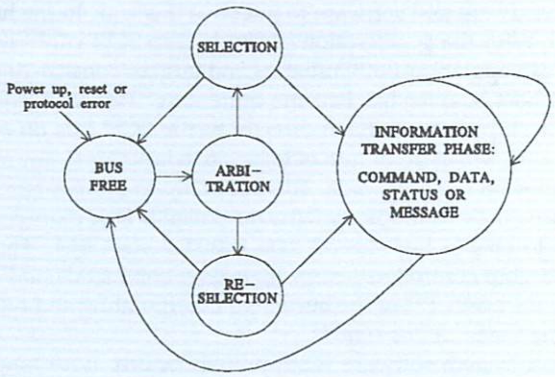
fer period determines the time between the end of one byte of data transmission and the beginning of another. (Figure 3 illustrates bus phase flow.)

SCSI-2 ENHANCEMENTS

Although the original SCSI spec published in 1986 (SCSI-1) was a great stride forward, it was lacking in some key areas. The SCSI-1 spec lacked sufficient definition to ensure compatibility between devices. The spec made reference to many different commands, but actually required the implementation of only a single one (REQUEST SENSE). The result was that different devices supported different commands, limiting the number of controllers that could "plug and play" in a particular SCSI system. Even as the SCSI-1 spec was being finalized this limitation was recognized, and the Common Command Set (CCS) group was formed to solve this problem by putting together an extended subset of the SCSI commands. The commands were extended to allow more detailed information to be sent to and from the device. A subset of all of the possible commands was chosen to make it easier for the peripheral device manufacturers. Limiting the number of commands that a SCSI device must respond to increases the chance that all of them can be implemented. Although the CCS document was not incorporated into the SCSI-1 specification, it's principles were sound and it was recognized as a de facto standard to be followed by SCSI peripheral manufacturers. This helped to reduce the amount of plug-and-play incompatibilities.

The CCS principles were incorporated into the SCSI-2 specification. SCSI-2 classifies commands as mandatory, optional, or vendor-unique. SCSI devices are required to implement at least all of the mandatory commands for their device types. Commands for direct access (disk drive), sequential-access

Figure 3: SCSI Bus Phases


SELECTION
Power up, reset or protocol error
BUS FREE
ARBI-TRATION
INFORMATION TRANSFER PHASE:
COMMAND, DATA, STATUS OR MESSAGE
RE-SELECTION

SCSI Signal Definitions

SCSI signals are carried on two separate cables. The A cable contains the 18 signals defined in the SCSI-1 spec, and the B cable contains the new definitions of SCSI-2, 29 in all. 11 signals are used for control, and 36 are used for data (including parity). (See Figures 4 and 5 for the conductor assigments for the A and B cables, respectively.)

REQ (request): Driven by a target to indicate a request for a REQ/ACK data-transfer handshake (for DB7-0).

ACK (acknowledge): Driven by an initiator to indicate an acknowledgment for a REQ/ACK data-transfer handshake (for DB0-7).

REQB (request): Driven by a target to indicate a request for a REQB/ACKB data-transfer handshake (for DB31-8).

ACKB (acknowledge): Driven by an initiator to indicate an acknowledgment for a REQB/ACKB data-transfer handshake (for DB31-8).

ATN (attention): Driven by an initiator.

BSY (busy): Wired OR signal driven by the initiator or target to indicate that the bus is being used.

C/D (control/data): Driven by a target that indicates whether CONTROL or DATA information is on the data bus.

DB(7-0,P) (data bus 7 to 0 and parity): On the A cable, these lines carry data back and forth between target and initiator. Also, each line is uniquely used by devices to announce their presence during bus arbitration.

DB(31-8,P1,P2,P3) (data bus 31 to 8 and parities): On the B cable, these lines carry data back and forth between target and initiator.

DIFFSENS (differential sense): Found only on a differential SCSI bus, this signal enables the differential drivers on the bus.

I/O (input/output): Driven by a target that controls the direction of data movement on the data bus (with respect to the initiator). Also used to distinguish between Selection and Reselection phases.

MSG (message): Driven by a target during the Message phase to indicate that the data contains message information.

RST (reset): Wired OR signal that can be set by any device that resets the bus. This is normally done at power-up or when a requested device does not respond.

SEL (select): Used during bus arbitration by an initiator to indicate that it wants to select a particular target, or by the target to reselect the initiator. The device to be selected is indicated on the data lines.

TERMPWR (terminator power): Provides power so that termination resistors can be positioned at both ends of the bus.

—GB

(tape drive), printer, processor, write-once (optical disk), CD-ROM devices, scanner devices, optical memory devices, medium changer devices ("juke box"), and communications devices are defined.

Another proposal of the CCS document that was included in SCSI-2 is the concept of reselection. In SCSI-1, when an initiator sent a command to a target it would occupy the bus until the target had completed the command. In SCSI-2, the initiator can disconnect from its target after a command has been issued. When the command is complete, the target arbitrates for the bus for the purpose of reselecting the device that sent it the command. It can then complete the operation by sending back data and status information. Now that the the SCSI initiator no longer has to wait for the SCSI target device to finish a command, it is free to send commands to other target devices so they can be executed concurrently.

This may be useful in a system that has more than one target SCSI device, but the more common situation is needing to send another command to the same target device. The SCSI-1 spec allowed only a single command to be outstanding from an initiator to a logical device of a target SCSI controller. In the case of disk drives this could be inefficient. Suppose, for example, a host CPU gets four separate requests from the operating system to read sectors that reside on tracks 1, 50, 2, and 52. If it can accept only one command at a time, the drive will waste a lot of seek time as it goes back and forth. If, however, all four commands can be sent at once, the controller at the disk can optimize the seek time by reading them in the order of 1, 2, 50, and 52. Because the host CPU speaks to the devices on the SCSI bus in terms of logical blocks of data, it has no idea where (or how) the data is stored on the device. Therefore, it is impossible for the host CPU to optimize the command sequence before it is sent to the SCSI device. This feature, called command queuing, has been adopted in SCSI-2. (Up to 256 commands can be queued.)

For a device to keep track of the many commands it queues up, a queue tag is assigned to each one, allowing each command to be uniquely accessed. Once a device has been selected on the SCSI bus and the IDENTIFY message is sent, a two-byte QUEUE TAG message is sent. The message consists of the desired queuing function and the identifier of the initiator. When a target controller reselects an initiator, the tag message is sent after the identifier. Commands sent without a queue tag are executed in the order received, but only one can be outstanding, as defined under SCSI-1 standards (this is to maintain compatibility between queuing and nonqueuing environments).

A third area of concern, SCSI-1's data-transfer speed, has also been improved in SCSI-2. SCSI-1's transfer rate maxed out at 5 MB/sec. This limitation was addressed from two different angles. The most obvious method to speed things up was to add more bits. Today's common 16- and 32-bit processors give the impression that SCSI-1's 8-bit bus is primitive. Consequently, the WIDE option was added in SCSI-2, adding 24 more data bits for a total of 32. The other method used to solve this problem was to speed things up. If you "wiggle" the lines twice as fast you will move data twice as fast—hence the definition of the FAST option of SCSI-2. ►

Figure 4: Conductor Assignments for a SCSI A Cable

Single-Ended Signal	Differential Signal	Cable Conductor Number		Differential Signal	Single-Ended Signal
GROUND	GROUND	1	2	GROUND	–DB(0)
GROUND	+DB(0)	3	4	–DB(0)	–DB(1)
GROUND	+DB(1)	5	6	–DB(1)	–DB(2)
GROUND	+DB(2)	7	8	–DB(2)	–DB(3)
GROUND	+DB(3)	9	10	–DB(3)	–DB(4)
GROUND	+DB(4)	11	12	–DB(4)	–DB(5)
GROUND	+DB(5)	13	14	–DB(5)	–DB(6)
GROUND	+DB(6)	15	16	–DB(6)	–DB(7)
GROUND	+DB(7)	17	18	–DB(7)	–DB(P)
GROUND	+DB(P)	19	20	–DB(P)	GROUND
GROUND	DIFFSENS	21	22	GROUND	GROUND
GROUND	GROUND	23	24	GROUND	GROUND
GROUND	TERMPWR	25	26	TERMPWR	TERMPWR
GROUND	GROUND	27	28	GROUND	GROUND
GROUND	+ATN	29	30	–ATN	GROUND
GROUND	GROUND	31	32	GROUND	–ATN
GROUND	+BSY	33	34	–BSY	GROUND
GROUND	+ACK	35	36	–ACK	–BSY
GROUND	+RST	37	38	–RST	–ACK
GROUND	+MSG	39	40	–MSG	–RST
GROUND	+SEL	41	42	–SEL	–MSG
GROUND	+C/D	43	44	–C/D	–SEL
GROUND	+REQ	45	46	–REQ	–C/D
GROUND	+I/O	47	48	–I/O	–REQ
GROUND	GROUND	49	50	GROUND	–I/O

Both methods of increasing the data throughput are not without their respective costs. Because all 50 pins of the SCSI-1 cable were already used, either the cable must grow or another cable must be introduced to add the additional data bits. Because SCSI-1 and SCSI-2 devices can co-exist in a system, increasing the size of the cable would make interconnections awkward. As a result, it was decided to add another cable. The original 50-pin SCSI cable is now known as the A cable, and the new 68-pin cable as the B cable. FAST SCSI has the limitation that it will work reliably only in a differential SCSI bus implementation. Because the timing is tightened during FAST operation, single-ended FAST SCSI cannot keep up. FAST SCSI also requires that all components in the data path must support the higher data rates.

The combination of FAST and WIDE SCSI provides a means to transfer data at a maximum rate of 40 MB/sec. This may seem impressive, but it's overkill for most of the common applications of SCSI. After all, a disk can spin only so fast! SCSI devices do contain RAM memory buffers, and depending on how much and how intelligently the device stores data from a disk in RAM will determine how effectively the bandwidth can be utilized. You must also take into account the speed at which the computer system itself can "swallow" the data. Generally, it is wasted effort to go to the greater cost and complexity of adding FAST or WIDE SCSI if the host system cannot keep up. As an example, the DMA bandwidth of the Zorro II system bus in an Amiga 2000 has a maximum bandwidth of about 3.5 MB/sec. Therefore, a SCSI host adapter connected to the Zorro II system bus of an Amiga cannot even keep up with SCSI-1. The DMA interface to the host adapter of an Amiga 3000, however, is located on the local 68030 bus and is capable of speeds up to 50 MB/sec.

COMMON ACCESS METHOD

The SCSI bus hardware and software may be well defined as pertains to the physical and logical interfacing between initiators and targets, but the interface between the SCSI host adapter in a computer system and the computer's operating system is not so clear. Because SCSI is implemented on many hardware platforms that have different system buses and dif- ►

Figure 5: Conductor Assignments for a SCSI B Cable

Single-Ended Signal	Differential Signal	Cable Conductor Number		Differential Signal	Single-Ended Signal
GROUND	GROUND	1	2	GROUND	GROUND
GROUND	+DB(8)	3	4	–DB(8)	–DB(8)
GROUND	+DB(9)	5	6	–DB(9)	–DB(9)
GROUND	+DB(10)	7	8	–DB(10)	–DB(10)
GROUND	+DB(11)	9	10	–DB(11)	–DB(11)
GROUND	+DB(12)	11	12	DB(12)	–DB(12)
GROUND	+DB(13)	13	14	–DB(13)	–DB(13)
GROUND	+DB(14)	15	16	–DB(14)	–DB(14)
GROUND	+DB(15)	17	18	–DB(15)	–DB(15)
GROUND	+DB(P1)	19	20	–DB(P1)	–DB(P1)
GROUND	+ACKB	21	22	–ACKB	–ACKB
GROUND	GROUND	23	24	DIFFSENS	GROUND
GROUND	+REQB	25	26	–REQB	–REQB
GROUND	+DB(16)	27	28	–DB(16)	–DB(16)
GROUND	+DB(17)	29	30	–DB(17)	–DB(17)
GROUND	+DB(18)	31	32	–DB(18)	–DB(18)
TERMPWRB	TERMPWRB	33	34	TERMPWRB	TERMPWR
TERMPWRB	TERMPWRB	35	36	TERMPWRB	TERMPWR
GROUND	+DB(19)	37	38	–DB(19)	–DB(19)
GROUND	+DB(20)	39	40	–DB(20)	–DB(20)
GROUND	+DB(21)	41	42	–DB(21)	–DB(21)
GROUND	+DB(22)	43	44	–DB(22)	–DB(22)
GROUND	+DB(23)	45	46	–DB(23)	–DB(23)
GROUND	+DB(P2)	47	48	–DB(P2)	–DB(P2)
GROUND	+DB(24)	49	50	–DB(24)	–DB(24)
GROUND	+DB(25)	51	52	–DB(25)	–DB(25)
GROUND	+DB(26)	53	54	–DB(26)	–DB(26)
GROUND	+DB(27)	55	56	–DB(27)	–DB(27)
GROUND	+DB(28)	57	58	–DB(28)	–DB(28)
GROUND	+DB(29)	59	60	–DB(29)	–DB(29)
GROUND	+DB(30)	61	62	–DB(30)	–DB(30)
GROUND	+DB(31)	63	64	–DB(31)	–DB(31)
GROUND	+DB(P3)	65	66	–DB(P3)	–DB(P3)
GROUND	GROUND	67	68	GROUND	GROUND

ferent hardware implementations of the SCSI host adapter, there is a great need to mask differences with a standard software interface. Also, some hardware platforms make no provisions for the SCSI host adapter to be used with different devices attached to it. In an effort to resolve these issues, an ad hoc industry committee called the Common Access Method (CAM) committee was formed. One goal of CAM is to provide similar software interfaces across hardware platforms, minimizing the effort involved in creating device-driver software for a new system.

Using the CAM approach, the operating system dependencies translate the system request into a CAM structure that is passed to the CAM model. Within the CAM module, the structure is translated into SCSI hardware instructions and a SCSI command. Upon completion of the command, status is then returned to the user application.

SCSI'S PHYSICAL REQUIREMENTS

The SCSI spec defines all the SCSI bus signals that need to pass between the devices. This signal information can be transferred via two methods: single-ended SCSI or differential SCSI. (A particular SCSI bus implementation is either one or the other.) The logic level of a single-ended signal is determined by the voltage of a single wire compared to the common ground. A differential signal requires two wires, and its level is determined by comparing one of the wire's potential to the other. While requiring more complex interface drivers, differential SCSI is less susceptible to electrical noise. According to the spec, differential SCSI can be used for cable lengths up to 25 meters and a single-ended implementation is good for up to six meters. While differential SCSI may be appropriate when implementing a local area network (LAN) via SCSI, its added cost and complexity is usually not warranted in a system with a single host adapter.

The SCSI spec requires a 50-conductor cable for the mandatory A cable and a 68-pin cable for the optional B cable. Allowable connectors for the A cable come in several shapes: low-density shielded (which looks like a Centronics printer connector), low-density nonshielded (.1"x.1" ribbon connector), and high-density shielded and nonshielded. The B cable connector may be only high-density shielded or high-density nonshielded. The high-density connectors were not defined for SCSI-1, but were necessary so that both connectors could fit on the back of even small peripherals that support the SCSI WIDE option.

Internal connections for all SCSI host adapters for the Amiga conform to the low-density nonshielded standard specified in SCSI. Connections to the outside world, however, do not follow the standard. Although the 25-pin cable used for external connections does handle all the signals required for a single-ended SCSI bus, 25 of the ground wires have been sacrificed in the interest of space. Because there are still six ground wires remaining, the cable does perform adequately. (As a side note, Apple also uses the 25-pin external connector.) When attaching peripheral devices that conform to the standard connector, you will need adapter cables.

To ensure that signal quality on the SCSI bus is maintained, termination of the bus is required at both ends. In general, all SCSI devices come with the resistor packs for termination installed. If the device is not at either end of the cable, you should remove the resistors. More than two sets of termination resistors on the bus may cause the interface to fail, because the line drivers are not designed to handle the extra current.

THE THIRD GENERATION

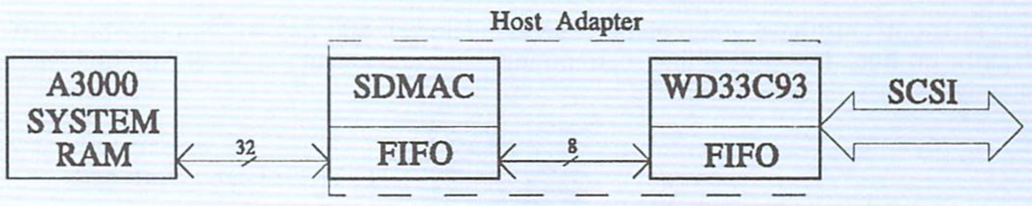
Although SCSI-2 has not yet officially been put to bed, work is already underway to define the new features to be adopted in SCSI-3.

Although the WIDE SCSI capability increased SCSI's bandwidth, the method in which it was implemented left many people unsatisfied. The issue of a single cable versus two separate cables was a hot debate while SCSI-2 was being defined, and it still is. A compromise proposes that the 68-pin B cable be used as the primary cable providing a 16-bit data path and that an optional second 68-pin cable provide another 16 bits of data if desired. You could connect both the 68-pin SCSI-3 and 50-pin SCSI-2 devices to the same cable, because there would be nine new signals added to each side of the 50-pin connector.

The ability to connect more than eight devices to the bus would be another nice addition. If the 68-pin cable becomes the standard, then eight more data lines will always be available for asserting device IDs, allowing up to 16 devices. Fast data rates and increasing numbers of devices on the cables promise to tax the bandwidth (and reliability) of single-ended SCSI buses. To counter this, a suggestion is before the committee that differential SCSI become the only option. Decreasing the specified maximum length of single-ended cables, however, may still make the 68-pin cable an option in compact systems.

As an alternative to electrical cable to connect SCSI devices, some folks have suggested that optical driver technology be used. Optical driver technology has the advantages of reducing bulky cable interconnections and dramatically increasing the physical distance between devices. This somewhat radical approach would require a dramatic departure ►

Figure 6: A3000 SCSI Implementation



from the current bus concepts and definitions, however, and it is more likely a candidate for definitions a generation or more beyond SCSI-3.

Another area being addressed is the concept of bus fairness. During bus arbitration, the device with the highest ID always wins out. This can leave devices with low IDs starving to use the bus. Although prioritization of devices is sometimes beneficial, some method needs to be provided to give the low priority guys a break once in a while.

Finally, a concept analogous to the Amiga's autoconfig may find its way into the SCSI definition—automatic configuration of SCSI bus IDs. No matter which suggestions make the cut, you can be sure the SCSI standard will keep growing with the times. ■

Greg Berlin is an eight-year veteran of the Commodore engineering staff. He was one of the principal designers of the A3000 hardware and is currently involved in the design of future high-end Amiga systems. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

The Amiga 3000 SCSI Host Adapter

The A3000 uses a single-chip SCSI controller, the WD33C93, to construct the SCSI host adapter. The WD33C93, from Western Digital, contains its own microcontroller, which greatly reduces the overhead required in the system to manipulate and monitor the SCSI interface. The WD33C93 pins connect directly to the SCSI bus to create a single-ended signal configuration.

The host CPU side of the interface is handled by the Super DMA Controller (SDMAC) gate array on the A3000's motherboard. This device allows data to be sent via DMA off the SCSI bus into memory. As data comes in from the SCSI bus, it is stored in a small internal byte-wide FIFO in the WD33C93. As this FIFO fills, the data is output to a 4-x-32-bit FIFO in the SDMAC chip. When its FIFO is full, the SDMAC requests the local bus from the 68030 and empties its FIFO into system RAM, 32 bits at a time. The bus is returned to the 68030 and the process is repeated until all the data has been received from the SCSI bus. (See Figure 6 for an illustration of the A3000 SCSI implementation.)

The WD33C93 was designed to support the SCSI-1 implementation; therefore, WIDE SCSI is not supported. Pin-compatible versions of the WD33C93 that support some additional characteristics of SCSI-2 are now available from Western Digital (WD33C93B). Although these devices do support FAST SCSI, it should not be used, since it requires a differential SCSI bus.

—GB

Who Helps Amiga Pros?

BIX -- the online service for people who know Amiga.

- ❑ Connect with the Commodore technical team
- ❑ Get quick answers to tough coding questions
- ❑ Interact with other Amiga Developers
- ❑ Download source code, utilities and other programs
- ❑ Keep up with the latest Amiga developments
- ❑ Send and receive private e-mail with binary attachments
- ❑ Chat with other Amiga pros in real time

All for just $39 for three months plus $3 per connect hour weeknights and weekends or $6 per connect hour weekdays via BT Tymnet.

Don't miss out! Just have your computer and modem call 800-225-4129 or 617-861-9767 and subscribe on-line. It's easy, at the Login prompt enter bix and at the Name? prompt enter bix.amiga.

BIX 800-227-2983 or 603-924-7681

On Disk

The Basics of Ray Tracing

Last issue's 3-D object viewer graduates to a higher level of sophistication.

By Brian Wagner

PHOTO-REALISM IS a popular catch phrase among computer graphics developers and users alike. One technique at the forefront of this explosion is ray tracing. From a programming standpoint, ray tracing is extremely elegant. With it, creating such dramatic effects as shadows, reflections, and even refractions (bending of light) is very easy.

To illustrate the basics, I'll walk you through a small program (called tracer) that generates images of 3-D objects using the ray-tracing rendering technique. Don't set your hopes too high: This is not a super-sophisticated ray tracer. I kept the example program simple, so the concepts would be as clear and understandable as possible.

Before we begin, you need to know a few things about how tracer works. The first consideration is the 3-D object format it represents. For this example, I chose the GEO format introduced by VideoScape 3-D (Oxxi/Aegis). Because it is an ASCII format, you can create and edit the objects with just about any text editor or word processor. (For a full description of the GEO object file format, refer either to the VideoScape 3-D documentation or the ReadMe file in the Wagner drawer of this issue's disk.) To describe how to view the object, we give tracer another ASCII file. This viewing-description file will contain the following information:

Where we are located
The position we are looking at
The fraction of the screen to use
Light source position
View window size
Position of the ground
Center of projection (eye location)

In practice, of course, it will contain coordinates in 3-D space and numeric settings instead of words. For example:

```
0 0 50
0 0 0
1.0
- 50 50 50
500 500
0
250
```

(For more details on the basic 3-D coordinate system—axes, planes, vectors, and so on—see "Building a 3-D Object Viewer," p. 15, June/July '91. In this article, we'll assume that the coordinate system is defined as shown in Figure 1.)

The final option we need to send tracer is the size of the screen to use. We'll use values between 1 and 4 to represent the following screen resolutions (sizes):

```
1 320x200
2 320x400
```

►

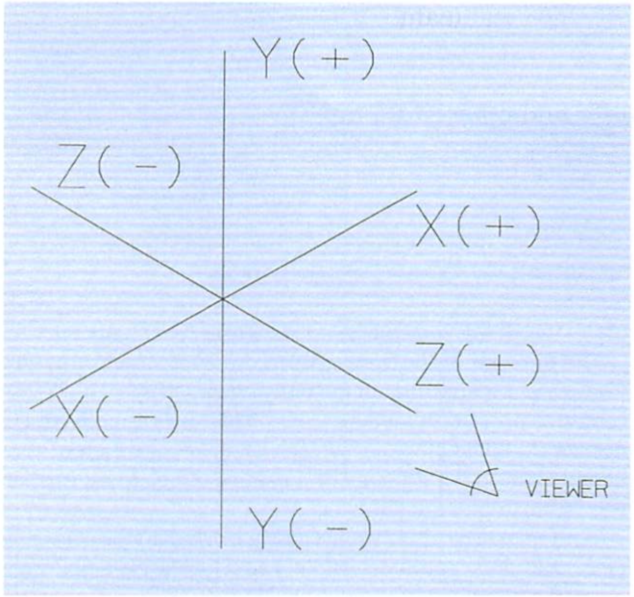


Figure 1: The 3-D coordinate system.

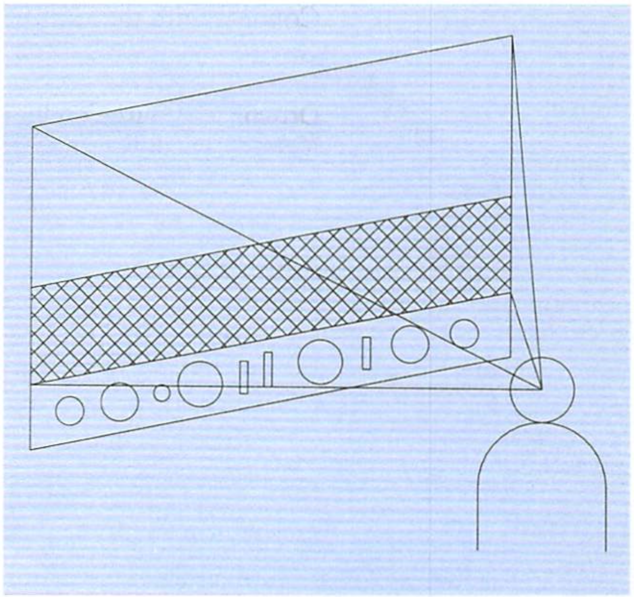


Figure 2: A pilot sees reflected light rays passing through his windshield.

3 640x200
4 640x400

You will actually have more control over the image size because one of the viewing-description file's parameters defines which fraction of the screen to use. In other words you can specify, for example, that the program render in only $^1/_2$ or $^1/_4$ of the screen.

Putting it all together, a sample tracer command line would look like:

```
tracer objectfile viewfile screenmode
```

If you want to skip ahead and look at the program now, feel free. Keep in mind, however, that the program itself only writes RGB files (.red, .grn, .blu), which must be converted into a picture for viewing. I included the public domain program View in the Wagner drawer for doing so. (Refer to View's documentation for more information on viewing RGB files.)

MODEL VIEWS

To make some sense of the values in the viewing-description file, let's define the model we will use to understand ray tracing. Imagine a pilot sitting in his cockpit looking out of the windshield at the world. The only reason the pilot sees anything at all is that light (most likely from the sun) is reflecting off the environment around him and then passing through the clear windshield into his eyes. (See Figure 2.)

Granted, this is an extremely basic description of how we "see." But, as you will discover, it is exactly the approach to take when ray tracing a picture. Now, imagine that the plane's windshield is actually your computer's screen. When you sit and use your computer, you are the pilot looking through/at your windshield/screen. If we use this relationship, you can view anything that is on the far side of the monitor. I know, that would mean that our object would have to be inside the monitor to be viewed through the screen. As this isn't too feasible, we simulate this situation inside the imaginary 3-D world of axes and planes. One way is to suppose that the windshield/screen is the XY plane, which

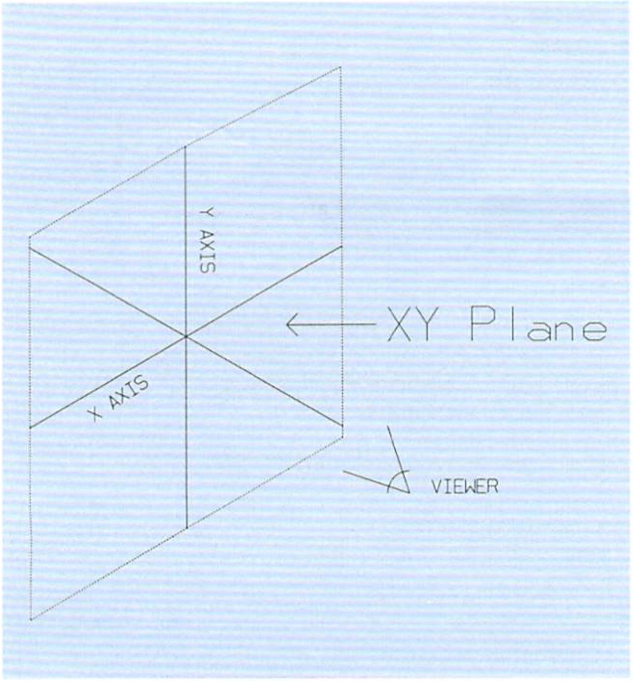


Figure 3: The windshield is the XY plane; the pilot is on the Z axis.

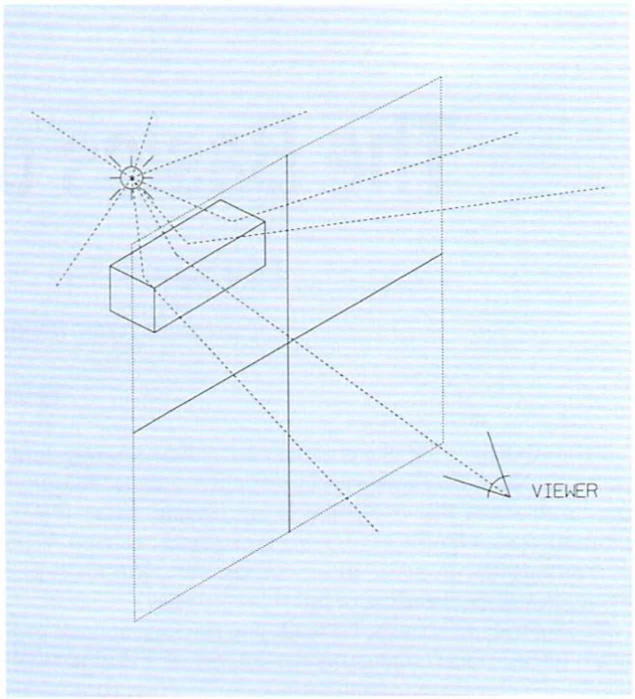


Figure 4: Following every ray from the light source wastes a lot of effort.

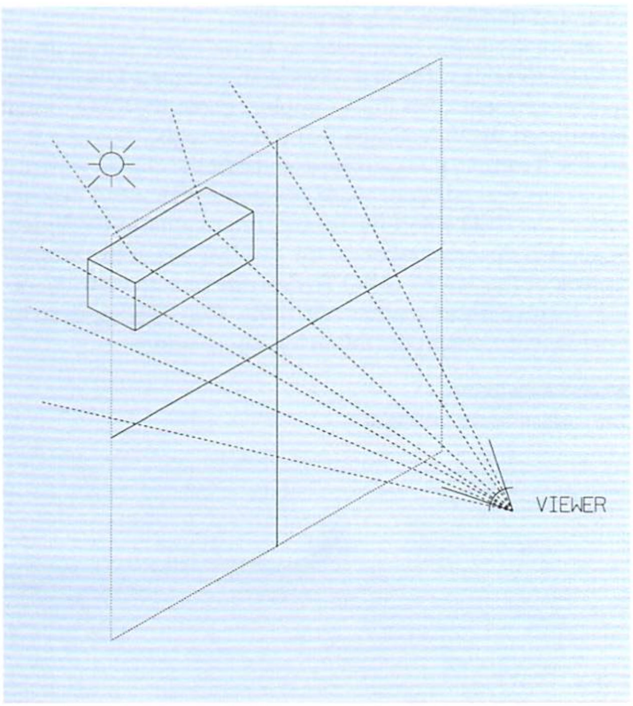


Figure 5: Following sight rays back to the source is more efficient.

places the viewer somewhere on the Z axis. (See Figure 3.)

The distance of the viewer from the XY plane is very important and is specified in the viewing-description file as the center of projection (eye location). This distance effectively defines the "field of view," which can make the difference between a flat-looking image or a fisheye effect. I will explain this value in more detail later.

With this scenario, any objects that existed on the *other* side of the XY plane could be visible. I say "could be visible" because the object is not necessarily "in view." (You'll see how this is determined a bit later.)

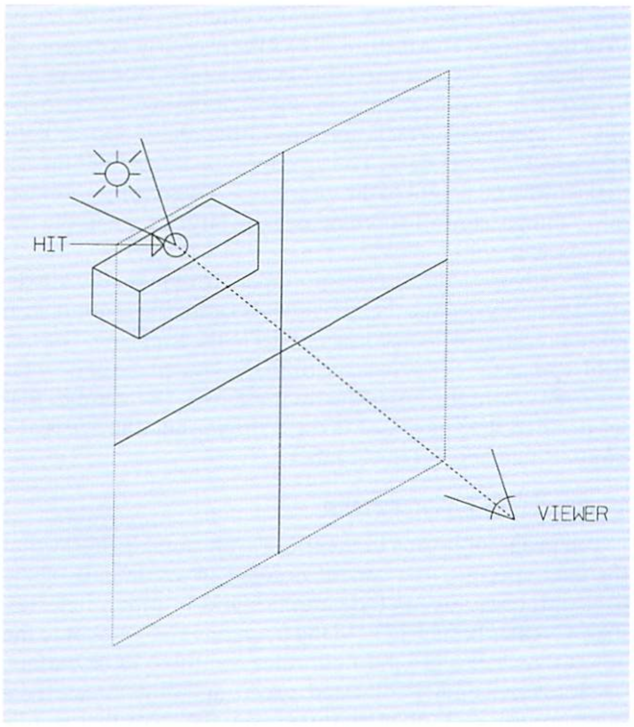


Figure 6: When you score a sight-ray hit, calculate the light reaching the object.

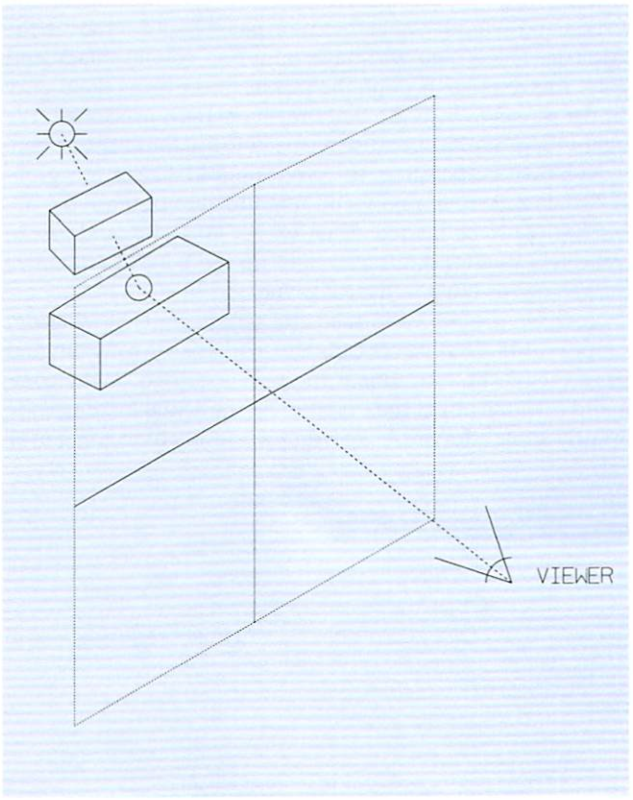


Figure 7: If the light source is blocked from the object, the object has a shadow.

Remember, light reaching the pilot's eyes defines what he sees. In our case, we'll have a light source, which is specified in the viewing-description file, to shed light on the object. If we tried to simulate the true physics of light, we would have to follow large numbers of light paths as they left the light source. First, we'd have to check if any of the paths hit the object, and then if any passed through the windshield/screen (or XY plane) to the viewer (from now on, I will refer to the XY plane as the "viewplane" and the viewer as the "eye"). It could take forever to chase random paths (rays) from the light source to see if they ever reached the eye, and only a small fraction of the light rays we followed would reach the eye at all. (See Figure 4.)

This is where we reverse nature's process. Computer graphics ray-tracing algorithms are actually backwards ray tracers. Instead of tracing light rays from the light source, we will trace "sight" rays from the eye (viewer) backwards, through the viewplane (XY plane), into the scene. (See Figure 5.) This way, we can determine exactly what is visible at any location on the viewplane. We will use the light source only when we actually hit the object with a ray. When we score a hit, we will calculate the amount of light that reaches the point on the object. (See Figure 6.)

Of all the trademarks that make ray tracing famous (shadows, reflections, and refractions), I will restrict the discussion to shadows, as they are the easiest to produce. Consider our example for determining the amount of light that reaches a particular spot on an object's surface. In the case of a shadow, light is blocked from the surface position being "shaded." To check for this, we test to see if anything lies between the surface point and the light source. If the light is blocked, the surface point gets no light and remains dark. That's it—instant shadows! (See Figure 7.)

The remaining element in the viewing-description file adds the final enhancements to our simple ray tracer—a ground and a sky. We will use the ground-level specification to describe where the ground lies. The sky will be rendered everywhere "above" the ground value. Although it may not sound like it right now, this really is a simple effect.

STRUCTURED COMMENTS

With the 3-D stage set, we are ready to examine the code. The first file, the header file tracer.h, contains all the structure definitions and a few constant definitions. For example, the Polygon structure below describes a single polygon:

```
struct Polygon {
   SHORT cnt;
   SHORT *vtx;
   FLOAT nx, ny, nz;
   SHORT r, g, b;
};
```

(Note: I assume that you are familiar with the concept of describing objects with polygons and vertices. If you are not, please refer to "Building a 3-D Object Viewer.")

The first field, cnt, is the number of vertices in the polygon. The second field, *vtx, is an array containing indices into the list of the object's vertices. The next three fields, nx, ny, and nz describe this particular polygon's surface normal. Surface normals are simply direction vectors that define which direction the polygon is facing. (We'll talk about this in more detail later.) The last three fields, r, g, and b define the color of the polygon as a mix of red, green, and blue. Each color value can range from 0 to 255.

The next structure is the Vertex structure, which simply contains the XYZ coordinates of a point in 3-D space:

```
struct Vertex {
   FLOAT x, y, z;
};
```

►

Keep in mind that these locations are used to define a polygon's shape.

We will use the third structure, Ray, to describe a ray.

```
struct Ray {
   FLOAT ox, oy, oz;
   FLOAT dx, dy, dz;
};
```

It is really quite simple; rays need to have only a starting point and a direction defined. The first three fields define the ray's starting point in 3-D space, and the last three fields describe the direction the ray travels from the starting point.

Following Ray is the Triangle structure:

```
struct Triangle {
   FLOAT x1, y1, z1;
   FLOAT x2, y2, z2;
   FLOAT x3, y3, z3;
   FLOAT nx, ny, nz;
};
```

As you will see later, we can efficiently check to see if only a triangle and ray intersect. Therefore, we will break the object's polygons into triangles before testing for intersections. The Triangle structure holds these intermediate triangles. The first nine values define each of the triangle's three vertices, while the last three values define the triangle's surface normal.

In turn, the Intersection structure holds information relating to the intersection of a ray and a polygon (or triangle) when one occurs:

```
struct Intersection {
   FLOAT ix, iy, iz;
   FLOAT dist;
   VOID *poly;
};
```

The first three values define the exact location in 3-D space of the intersection. The distance from the eye to the intersection point is stored in dist. The final field is a pointer to the Polygon structure of the polygon that was hit.

The three values in the Color structure:

```
struct Color {
   SHORT r, g, b;
};
```

define the polygon's color as described earlier.

Finally, the ViewOpts structure holds all the information in the viewing-description file:

```
struct ViewOpts {
   FLOAT cax, cay, caz;
   FLOAT lpx, lpy, lpz;
   FLOAT scl;
   FLOAT lsx, lsy, lsz;
   FLOAT vpx, vpy;
   FLOAT wdy;
   FLOAT cpd;
};
```

The first three fields define where the eye is located, and the following three define the position in 3-D space at which the eye is looking. The scl field describes which fraction of the screen to use. We'll use this value to scale the width and height of the screen mode specified in the command line, letting us create images that cover, for example, only $^1/_2$, $^1/_4$, or $^1/_8$ of the screen. This feature allows the user to generate quick test images. The next three fields define the location of the light source. The location of the light source in relation to the object determines which portions of the object appear light and which appear dark. The vpx and vpy fields hold the actual width and height of the viewplane. These values, among other things, control zooming or how large or small the object appears.

The wdy field houses the ground's position on the Y axis. We need only a single height value because the ground is an infinitely large plate that lies parallel to the XZ plane. The last field, cpd, is the eye's distance from the viewplane or how far the eye is from the XY plane on the Z axis. (Remember that we decided the viewplane is the XY plane and the eye is at a point somewhere on the Z axis.)

TYING IT ALL TOGETHER

The program begins in the tracer.c file. First it checks that the user specified the correct number of arguments. If the user did, tracer opens the Graphics and Intuition libraries, then allocates buffers to hold the polygons and vertices of the object. Next, the program allocates the three RGB buffers, one for each primary color, red, green, and blue. As we trace rays through each pixel on the screen, the color that results will be stored in these three buffers.

Following all buffer allocations, the program opens the screen and window, sets the color palette, and removes the title bar. The screen will have only two colors—black and gray. As we trace through the pixels, tracer turns them gray on the screen to indicate that they have been processed. We remove the title bar because we need to see the entire screen.

As a final preparation, the program initializes a few global variables. We set npoly and nvert to zero because the object has yet to be loaded. Then we set gnx, gny, and gnz to indicate a vector pointing straight up the Y axis to initialize the ground as lying in the XZ plane.

Now we are ready to load the object- and viewing-description file. First tracer calls the loadobject() function and passes it the first argument the user specified in the command line (the object file's name). As the actual loading code is fairly straightforward, we won't waste any time going over

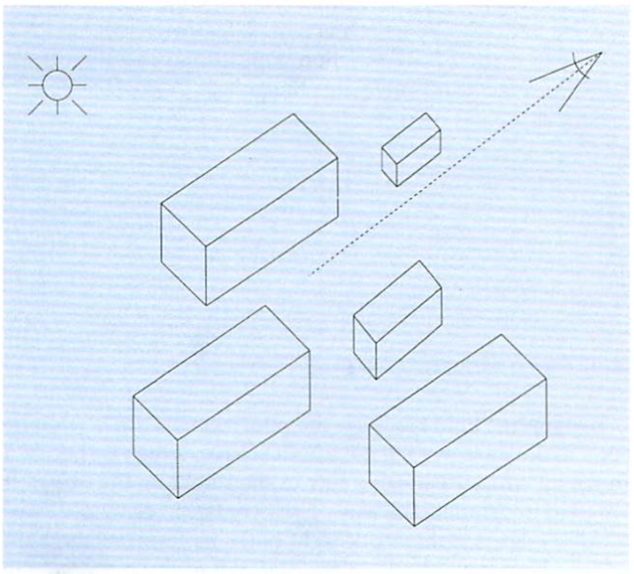


Figure 8: Your viewer can by anywhere in 3-D space. To display the view, rotate the scene.

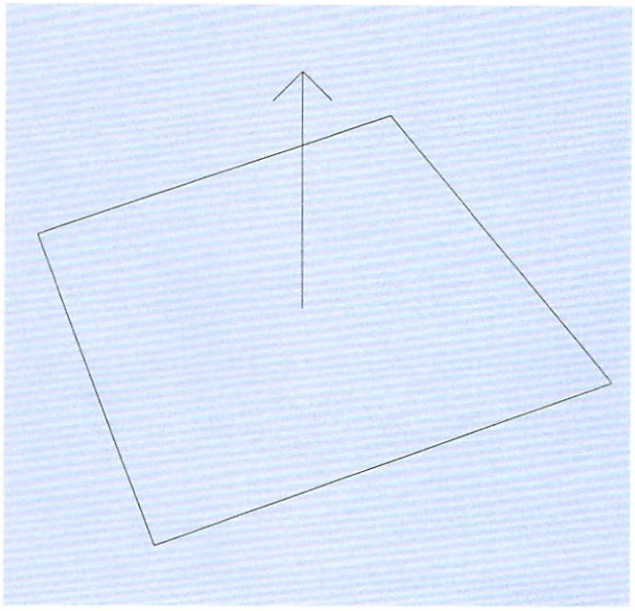


Figure 9: The surface normal of a polygon face.

it. The program loads the viewing-description file and places it in the global vopts structure by calling loadvopts().

With everything opened, allocated, and loaded we can begin the rendering process, which is broken down into four major functions. The first to be called is transform() (located in math.c). In it, the object, light source, and ground normal are all transformed into the viewplane coordinate system. This is a little tricky, so pay close attention. Remember that we are using the XY plane as the viewplane, but you can specify where the eye is located in 3-D space, and it could be looking at any other location in 3-D space. (See Figure 8.)

If we always use the XY plane as the viewplane, we need to rotate/move the object, light source, and ground normal to get the same view of the object from the XY plane as we would from the arbitrary location in 3-D space. If you don't fully understand this concept, don't worry. It should become more clear as we continue. Keep in mind that this operation greatly simplifies the actual ray-tracing process. (We won't dig into the actual details of this function, as you can find a detailed explanation in "Building a 3-D Object Viewer.")

After the transformation is complete, tracer calls calcnormals() (in math.c). This function steps through each of the object's polygons and calculates the polygon's surface normal using the vector cross-product operation. (Again, the details of this process can be found in the previous article. For now, it is important to know only that surface normals are simply direction vectors that indicate the direction in which a polygon faces. See Figure 9.)

FIND THAT RAY

Let's begin ray tracing by calling the traceimage() function (located in image.c). Notice that tracer passes this function a pointer to our window's rastport. We will need this to turn the pixels gray as they are traced. The rather complex formula that leads off the function:

```
ar = (scrw * ((scrh * 4.0) / (scrw * 3.0))) / scrh;
```

calculates a scaling value used to compensate for the fact that monitors are not square. This difference in width and height is indicated by the aspect ratio. The above formula uses an aspect ratio of $^4/_3$, which is common to most monitors.

Next, the program moves the width and height of the viewplane out of the global vopts structure and into the local vpx and vpy variables:

```
vpx = vopts.vpx * ar;
vpy = vopts.vpy;
```

Notice that the width is scaled by the aspect ratio value. The width and height values (which originally come out of the viewing description file) are very important. When we start tracing, we are going to shoot rays through each pixel on the screen out into 3-D space. Therefore, we need to relate the screen size (in pixels) to an equivalent in the 3-D world. The exact values used are actually dependent on the size of the numbers that describe the object.

As a general rule, the width and height should be equal (as specified in the viewing-description file) and about $^1/_2$ the largest dimension of the object. For example, if the largest object dimension were 1000, a good width and height would be 500. These values can, of course, be larger or smaller. In fact, you can use them to zoom the final image in or out. Smaller values will make things appear larger, while larger values will make things look smaller.

The next two lines calculate exactly how many pixels will be traced:

```
actw = scrw * vopts.scl;
acth = scrh * vopts.scl;
```

Here we use that scaling value from the viewing-description file to scale the original width and height of the screen (in pixels). Finally, we set the drawing color to pen number one—gray.

We begin tracing by starting two loops, one for the width and the other for height. This will allow us to cover the specified range of pixels by working top to bottom, left to right. Each pixel is handled on an individual basis. The beauty of the ray-tracing algorithm is that it deals with each pixel, one at a time. Instead of examining the object's polygons and figuring out which pixels to set to which color, the program goes backwards and looks at the pixel first.

We want to create a ray that starts at the eye and passes through each pixel. Because our screen full of pixels is represented in 3-D space by the XY plane, this is very simple:

```
px = (FLOAT)i + 0.5;
px = vpx * ((px / actw) – 0.5);

py = (FLOAT)(acth – 1 – j) – 0.5;
py = vpy * ((py / acth) – 0.5);
```

The first two lines find the horizontal, or X-axis, position. The last two find the vertical, or Y-axis, position. Because they are very similar, we will examine the first two lines only.

```
px = (FLOAT)i + 0.5;
```

adds 0.5 to the pixel's X coordinate because we want to deal with the center of the pixel.

```
px = vpx * ((px / actw) – 0.5);
```

converts the pixel location into the range of 0.0 to 1.0 by dividing the pixel's X coordinate by the total number of pixels to be traced horizontally. Subtracting 0.5 from this value ensures that it falls in the range of – 0.5 to 0.5. Finally, we multiply by the viewplane width (stored in vpx). ►

The same process produces the Y-axis position in 3-D space. The difference is that the position is reversed because vertical screen coordinates are reversed from those in 3-D space.

With the pixel's location in 3-D space stored in px and py, let's build a ray. First, we calculate it's origin. Simple: The origin is the eye position and the eye lies on the Z axis. Therefore, the X and Y coordinates are both zero. The Z coordinate is set to the center of projection (eye location) specified in the viewing-description file. Remember, because the viewplane is at the center of the Z axis (0), this value specifies distance from the viewplane. (See Figure 10.)

With the origin set, we calculate the ray's direction. Because the direction will be from the ray origin through the previously calculated pixel location, we create a vector by subtracting the ray's origin from the pixel location. After converting this vector into a unit vector (a vector with a length of 1), we store the ray direction in the Ray structure.

FOLLOW THAT LIGHT

Now that we have a ray, we can "trace" it and see what, if anything, it hits. First, the program initializes the intersection structure, isec, to indicate that there is currently no hit. Then it loops through all of the polygons and, by calling the polygonhit() function (in math.h), tests to see if the ray intersects any of them.

At the heart of every ray tracer is the code that tests for intersections between a ray and some shape—in our case, a triangle. Because our polygons can be larger than triangles, however, we must also test the polygon in triangular pieces for intersection. The polygonhit() function does the necessary break-ups and calls the trianglehit() function (also in math.h) to check for intersection. (See Figure 11.)

The code that performs the triangle/ray intersection test is quite complicated, so we will go over it on a basic level only (study it at your own pace). First, the ray is checked to see if it intersects the plane in which the triangle lies. If so, the routine checks to see if the intersection point is either behind the ray origin or if it is beyond an intersection already found. Remember, we want to end up with the closest intersection. If the intersection is found to be "good," the routine concludes by determining whether this intersection point actually lies inside the triangle. If the answer is yes, the intersection point and distance is stored in the supplied intersection structure and the function returns.

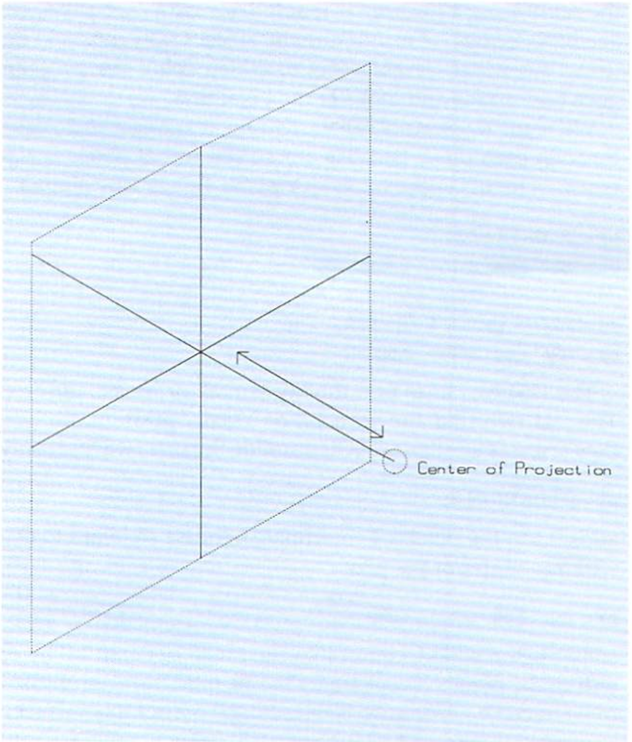


Figure 10: The center of projection specifies the eye location and its distance from the viewplane.

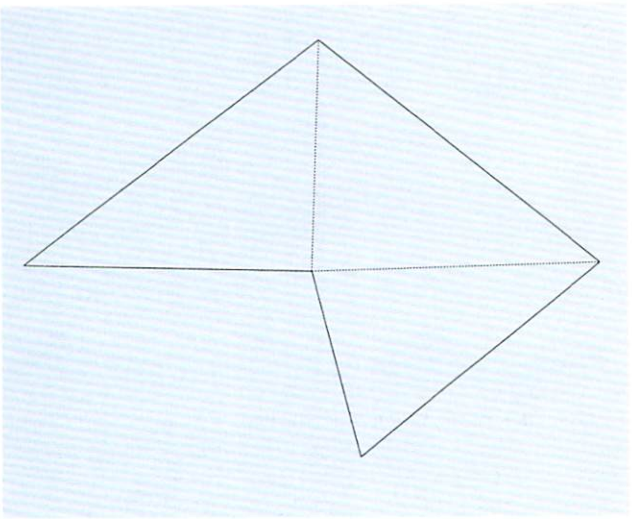


Figure 11: Break a polygon into triangles before testing for ray intersections.

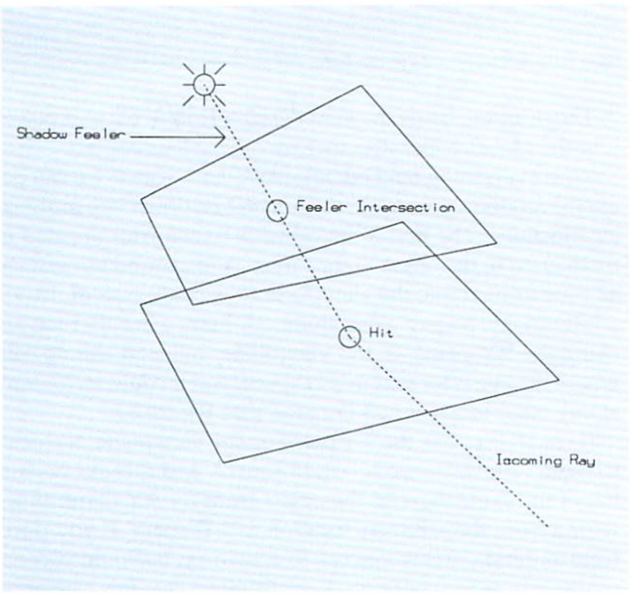
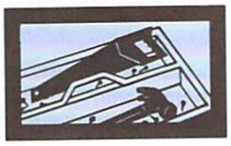


Figure 12: Check for a ground hit, then any objects that block the light.

All the polygons are checked in this manner. After the ray is checked for intersection with all the polygons, it is checked to see if it intersects the ground, via the groundhit() function (in math.h). This function is really the same as the first part of trianglehit(). The ray needs to be checked only to see if it intersects the XZ plane at the Y-axis location specified in the viewing-description file. In addition, groundhit() returns true only if the intersection is closer than any of the triangle/ray intersections.

If a ground hit is found, the ground will appear at that pixel, so it needs to be shaded. The first step in this process is to check if this point lies in shadow. To test this, we call shad-

Continued on p. 63



TNT

Technical News and Tools from the Amiga Community.

Compiled by Linda Barrett Laflamme

Official Words from CATS

In response to the often-heard statement "What this market needs is consistency across applications" comes the ***Amiga User Interface Style Guide***. Written by CATS, it describes how an application's user interface should look and operate, setting forth Commodore's interface standards for the Workbench, the Shell, and ARexx. The guidelines are backed by definitions, descriptions, and illustrations of each of the interfaces. As only behavioral guidelines are presented, both creatively and technically minded interface designers can benefit. If you can't wait to find a bookstore, you can order the *Amiga User Interface Style Guide* directly from Addison-Wesley Publishing Company (Jacob Way, Reading, MA 01867, 800/447-2226) for $21.95 ($28.95 in Canada).

For hardware-minded registered developers, CATS is offering an **Amiga 3000 schematics manual** for $35 (plus $2.50 for Canadian shipping, $5.00 for overseas). Direct your requests to CATS Orders, 1800 Wilson Dr., West Chester, PA 19380 and ask for part number 314677-02.

Be a Winner

Want to win $10,000? If you are working on (or have finished) a program or device that would benefit the physically or learning disabled, here's your chance.

Johns Hopkins University is conducting a National Search for Computing Applications to Assist Persons with Disabilities (CAPD). Sponsored by the National Science Foundation and MCI Communications Corp., the **National Search** is a competition for ideas, systems, devices, and software that are creative and affordable solutions to the problems faced by the disabled. The grand prize for the best computer program or device is $10,000, and over 100 additional prizes will be awarded. Regional winners will be selected in December at science museum exhibitions across the country.

The top 30 regional winners will be invited (all expenses paid for the top 10) to exhibit at The National Exhibition at the Smithsonian Institution in Washington, DC, on February 1 and 2, 1992. There the grand prize winner and top 10 national winners will be honored.

Entry fliers are available from CAPD, PO Box 1200, Laurel, MD 20723. The deadline for entries is August 23, 1991, but don't let the short timeframe deter you. National Search officials promise to expedite last-minute entries by fax or other means when necessary.

Optical Fish

Compatible with CDTV and ISO-9660-type CD-ROM drives, the **Fred Fish Collection on CD-ROM** disc houses 410 floppies worth of the Fred Fish software library. The collection is provided both in its original form organized by disk and, as a bonus for BBS sysops, compressed by disk into .zip files. Several tools (including SID and PKAZIP) are included in the disc's root directory to facilitate file management. Available from HyperMedia Concepts Inc. (PO Box 85303, Racine, WI 53408, 414/632-3766), the Fred Fish Collection on CD-ROM sells for $69.95 (plus $2.00 shipping U.S., $10 foreign). As the collection expands, registered users may purchase updates every four months for $29.95 each. If you prefer to plan ahead, consider a subscription: $109.95 for the original disc and two updates. If you are a registered Amiga developer, call HyperMedia Concepts about their discount plan.

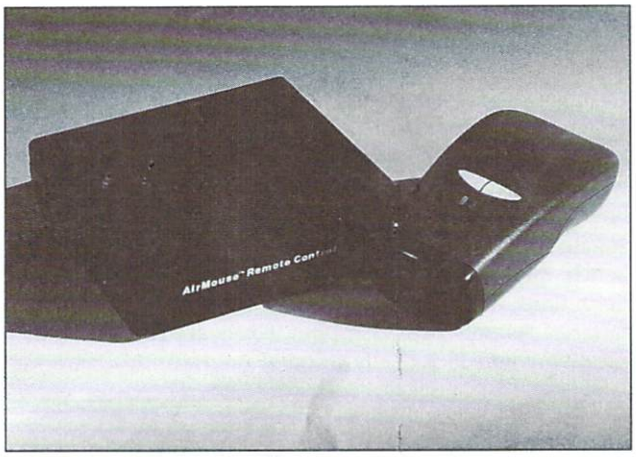
Faster, Faster

Great Valley Products wants to speed up your Amiga with its **Series II accelerator systems**. Topping the line, the GVP A3050 ($2999) is a 50 MHz 68030 accelerator board that comes populated with 4MB of 32-bit memory (expandable to 32MB), and a SCSI controller. Available optional drives are the Maxtor one-inch tall 120MB model or a half-height 340MB model. GVP also offers 22 MHz and 33 MHz 68030 models. The 22 MHz board comes standard with 1MB of RAM and can be ►

expanded to 13MB, while the 33 MHz version is shipped with 4MB of RAM and can be expanded to 16MB. For complete pricing, contact Great Valley Products, 600 Clark Ave., King of Prussia, PA 19406, 215/337-8770.

Where's the Tail?

Tired of the same old input devices? Consider the **AirMouse Remote Control** ($595), a bidirectional infrared, point-and-click input device. Designed with presentation software and CDTV titles in mind, the two-button remote unit will work up to seven meters from its basestation, which connects to your CPU via the serial port. Current drivers support OS 1.3 and CDTV, but a 2.0 driver is on the way.



Point and click from seven meters away.

Should you have a special implementation in mind, check out the Developer ToolKit. For $4995 you receive a developer-style AirMouse remote and basestation, plus drivers for the Amiga, MS-DOS, Windows 3.0, and Macintosh 6.0.X. If you have any problems, Selectech provides developer services. Contact Selectech Ltd., 30 Mountain View, Colchester, VT 05446, 802/655-9600.

Cheat Sheets

You went to the trouble of programming keyboard equivalents for menus choices, but now your users complain they can't remember the commands. Rather than hire a larger technical-support staff, consider K/B Optimizer **custom keyboard templates** and **reference cards**. Templates and cards are constructed of high-impact, UV-rated polypropylene plastic and coated with a waterproof, scratch-resistant plastic matte finish to elimnate glare. If none of their standard configurations will do, K/B promises to custom design templates and cards to your specifications. Pricing depends on quantity, ranging from $24 apiece for 10 templates ($29.95 apiece for 10 cards) to $5.35 each for 1000 templates ($9.95 each for cards). For exact prices and more details, contact K/B Optimizer, PO Box 877, Gardiner, MT 59030 406/848-7320.

Faster and Sharper

If the standard Amiga display modes aren't enough for your software, consider Digital Micronics Inc.'s 60 MHz **DMI010** ($1095) and **DMI020** ($1995) **graphics processor boards** for the A2000, A2500, and A3000. Both offer programmable control up to their maximum resolutions (1024x800 for the DMI010, 1280x1024 for the DMI020). The DMI010's eight-bitplane system uses a 24-bit, 16-million-color palette with 256 active colors and over 800,000 available pixels, while the DMI020's 24-bitplane system offers 16 million colors and over 1.3 million pixels. If that's still not enough power, the DMI010DB ($195) memory buffer option for the DMI010 promises to increase display speed or up resolution to 1280x1024. The DMI020's memory buffer option (DMI020DB, $595) doubles the available VRAM to speed up screen display. For further information, contact Digital Micronics Inc., 5674-P El Camino Real, Carlsbad, CA 92008, 619/931-8554.

Pass the Word

Run across any hot products, PD software, or news? We want to here about it. Send the details to TNT, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.* ■

On Disk

The AmigaWorld Tech Journal Disk

This nonbootable disk is divided into two main directories, *Articles* and *Applications*. Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas, not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

With the exception of the 2.0 include files (which are distributable only under license), the supplied files are freely distributable. Do not, however, resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

On Disk

RxTools

Where ARexx meets Intuition.

By Eric Giguere

AREXX, THE INCREASINGLY popular interprocess-control protocol that Commodore adopted under AmigaDOS 2.0, is a faithful Amiga implementation of the powerful REXX language. Because REXX was designed for console-oriented systems—not graphical user interfaces like the Amiga's—it comes as no surprise that ARexx lacks windowing and menuing functions. Thanks to ARexx's expandability, however, you can add Intuition-like capabilities to your ARexx scripts.

There are two ways of adding functions to ARexx. The most common method is to use an ARexx function library, which is an Amiga shared library that ARexx can access. (The ARexx mathematics library, RexxMathLib, is an example of a function library.) An often easier method is to use an ARexx function host. A function host is simply an ARexx-compatible application program capable of processing special messages from ARexx. TTR Development's RxTools package is such a program.

The RxTools software comes on a single bootable disk that includes not only the host, but also an extensive set of tutorial files and a simple RxTools script. To use RxTools, you need either Workbench 2.0 or Workbench 1.3 along with the ARexx software purchased from William Hawes. You should have at least one megabyte of memory, as the function host itself is almost 400K in size. A hard disk is also recommended. If you wish to share your RxTools scripts with others, they too must buy the package.

To access RxTools' functions, you must run the rx_tools.exe program in the background. RxTools provides an initialization sequence that your ARexx scripts can use to find out whether the function host is running, and to start it if not. The initialization sequence also adds the function host to the ARexx Library List (the list of all function libraries and function hosts that are active) so that ARexx will send it a message when it needs to locate a function.

BUILDING WITH OBJECTS

RxTools is based on the principles of object-oriented programming (OOP). There are many books that describe OOP in detail, but here's a quick primer: The basic entities in OOP are known as objects. In RxTools, for example, a window is an object. Objects have certain attributes; a window has a size and a position, for instance. Objects also allow you to modify those attributes through various methods; RxTools provides a method to move a window, for example. A method can be thought of as a procedure or function call, but one that is intimately associated with—actually a part of—an object.

Methods and attributes are not very interesting without another OOP feature known as class inheritance. Each object is a member of a class of objects. All objects in a class share attributes and methods, though of course each object has its own data space. For example, two windows both have a movement method, and both have size and position attributes, but the values of the two sets of attributes may be different. You can build new classes out of old classes simply by adding or modifying attributes and methods. Each child class inherits the attributes and methods of its parent class unless the new class's definition specifically overrides them. In RxTools, for instance, the console class is a child of the window class. Hence, a console object has all the capabilities of a window object plus a set of methods specific to creating and using an Amiga console device. Classes also have methods, but these are used mostly to create new objects.

Although object-oriented programming seems strange at first, if you play around with it for a while, you will likely find it to be quite elegant. To be truly elegant, however, OOP requires a programming language designed with objects in mind. While ARexx is not an object-oriented language, RxTools manages to get around that restriction. RxTools includes a reasonable set of object classes: menus, various classes of windows and consoles (including a functional text editor), file requesters, and some basic gadgets.

Let's dissect a simple RxTools script to get a better feel for the program. All RxTools scripts start with the following statement:

```
/* An RxTools Script */
interpret getclip( 'rx_tools_init' )
```

This rather strange statement does two things: It retrieves a string from the ARexx Clip List and then uses the INTERPRET instruction to execute the contents of the string as if they had been typed into the script. The string it retrieves must have been previously installed in the Clip List by executing the rx_tools.rexx ARexx script, which, as you will recall, also starts the RxTools function host. The string holds a series of ARexx statements that perform various initialization tasks, including ensuring that the RxTools function host is still alive (in case a previous script has terminated it), opening a notification port, and assigning useful values to a set of variables.

The next step is to open a window by calling the RxTools function host:

```
my_window _send('rx_console',_OPEN,
    10, 30, ,250, 80, 'My Window' )
```

(Keep in mind that if you want to split an ARexx statement across two or more lines, each line except the last one must end with a comma. Such commas are skipped by the interpreter, so the example above really has seven parameters, not eight.)

The _send() function is an RxTools function that "sends a message" to a class. In RxTools terms, sending a message to a class or object means invoking a method on that class or object. Do not confuse this with the Amiga's built-in interprocess communication (IPC) facilities. In this case, the _send() function is invoking the _OPEN method of the rx_console class, with four integers and a string as parameters for that method. This will create (open) a window of class rx_console at location (10,30), width 250, height 80, and title "My Window" on the Workbench screen. The returned string is a handle identifying the newly created window.

Once the window has been opened, you can display a string of text in it by using the following function call:

```
call send my_window, _PUT_STRING,
  'Hello world'
```

(The call instruction is used to invoke this function because the return value is unimportant.) Notice how this time we use the send() function instead of _send(). The leading underscore makes a difference: _send() invokes a method on a class, whereas send() invokes a method on an object. This particular send() sends a _PUT_STRING message to the window object just created. PUT_STRING takes a string and outputs it at the current cursor location.

Now we come across another of those interesting INTERPRET statements:

```
interpret getclip( 'rx_tools_event_handler' )
```

This retrieves and executes another string from the Clip List. This particular string waits for messages (of the IPC type) to arrive at the script's notification port, calls a function in the script, and then waits for another message to arrive. Any statements following the event handler (the INTERPRET statement) will not be executed unless they are part of an ARexx function.

What kind of messages does the notification port receive? It depends on the objects that the script defines. For example, most windows send a CLOSEWINDOW message when the user clicks the window's close gadget. The event handler receives this message and immediately calls the CLOSEWINDOW() function, which might be defined as follows:

```
CLOSEWINDOW: procedure expose packet
  parse arg the_window

  call send the_window, _CLOSE
  call send the_window, _DELETE
  call reply packet, 0
  exit 0
```

This routine closes and deletes the window that received the close event, and then terminates the ARexx script. (A script that opens several windows would include code to exit only if all windows have been closed.) Most object definitions include, as a parameter, the name of a function that is to be called when an event for that object occurs. For example, you could add a string gadget to the window using the statement:

```
call send my_window, _ADD_STRING_
  GADGET, 'StringGadget', ,10, 10, 100
```

When the user enters a string in the gadget and presses RETURN, the event handler receives an event and calls the StringGadget() function, which can then retrieve the string from the gadget and process it. An RxTools program is basically just a collection of ARexx functions that get called whenever events occur.

That, in a nutshell, is the skeleton of an RxTools program: Initialize RxTools, define and initialize all the objects, define the functions that the objects call, and then start the event loop.

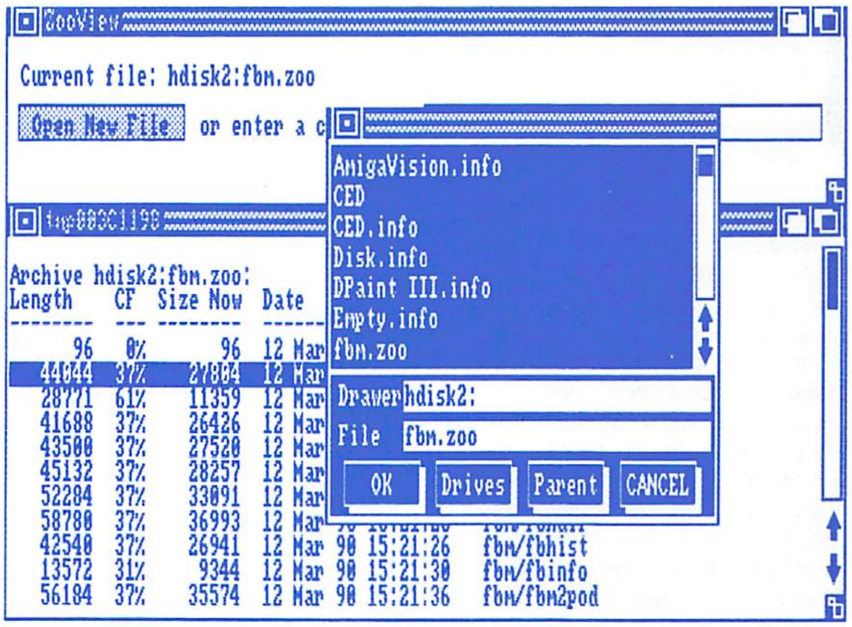
CLIMBING AND BROWSING

The set of all classes in an OOP environment forms what is known as a class hierarchy, or tree, because every class is ultimately a descendent of some root class. This hierarchy is important because it lets the programmer quickly determine how the various classes are related. You will find a class tree at the back of the RxTools documentation.

A good OOP environment includes a class browser, a tool that allows the programmer to traverse the class tree and view specifics about each class. The RxTools browser shows the inheritance path of a class and the methods that class and its objects support.

The browser is the most powerful tool you can use to write RxTools scripts because its information is complete and up-to-date. The only problem with the browser as it now stands is that there is no simple way to capture its output. Because of the way it was implemented, all its output is generated by the function host itself and hence appears only on the console where the function host was started. This limitation is a bug that I hope will be fixed for the next update.

RxTools has other problems as well, the most serious of which is its documentation. Not only is the 90-page manual meager and incomplete, but it ►



```
Current file: hdisk2:fbm.zoo
```

```
AmigaVision.info
CED
CED.info
Disk.info
DPaint III.info
Empty.info
fbm.zoo
```

```
Drawer hdisk2:
File   fbm.zoo
```

```
Archive hdisk2:fbm.zoo:
Length   CF  Size Now  Date
-------- --- --------  ------
      96  0%       96  12 Mar
   44044 37%    27804  12 Mar
   28771 61%    11359  12 Mar
   41688 37%    26426  12 Mar
   43500 37%    27520  12 Mar
   45132 37%    28257  12 Mar
   52284 37%    33091  12 Mar
   58780 37%    36993  12 Mar
   42540 37%    26941  12 Mar 90 15:21:26   fbm/fbhist
   13572 31%     9344  12 Mar 90 15:21:30   fbm/fbinfo
   56184 37%    35574  12 Mar 90 15:21:36   fbm/fbm2pod
```

An Intuition interface for Zoo by RxTools.

is poorly organized, full of errors, and lacks an index. The explanations need to be expanded to describe in detail what a function host is and how RxTools does its work. My recommendation: Once you have installed and started RxTools, leave the manual in its box. Use the tutorial examples and the class browser as learning tools instead.

Installing RxTools is also troublesome. The installation procedure assumes you want RxTools installed on your SYS: volume, a mistake if the partition is almost full. An installation script written in ARexx would be more flexible, and the ultimate goal should be an installation script that uses RxTools to install itself. Another problem is that RxTools does not always clean up after itself. It is not uncommon for an ARexx program under development to encounter an error and terminate. Unfortunately, RxTools does not free the program's resources, and so its windows and screens hang around until RxTools notices there is no ARexx program to notify. Even when RxTools is told to terminate, orphaned screens are often left behind to eat up valuable chip memory. Resource tracking needs to improve.

RxTools' error messages are annoying. Warning and minor error messages are displayed in system requesters, and, under 1.3, the buttons on the requester overlap with the last line of text. Serious errors are announced using the red-on-black alert requesters, which seem incredibly rude (and reminiscent of a GURU message) when you consider that at most the RxTools function host will crash. (This is not an unlikely occurence, either. I found that some of the RxTools tutorial scripts crash the function host.)

RxTools could use a few more objects, including list boxes and horizontal scroll bars. I would also like to see some support for the new 2.0 gadgets, windows, and screens—even if those features are not available to 1.3 users.

Finally, I would like to see RxTools decreased in size, or at least have access to a smaller, nonbrowsing version. The current size almost certainly precludes its use in developing commercial products.

FUTURE DIRECTIONS

RxTools is an interesting product that is almost crippled by bad documentation and bugs. That's a shame, considering how much work has gone into the design and implementation of the function host. To TTR's credit, however, the company has been quick to answer questions on BIX and has a BBS (608/277-8072) for making updates and bug fixes available to users. TTR is planning to fix many of RxTools' problems, and to revamp the manual. At this stage, however, it's hard to recommend RxTools to the novice ARexx user. It would be wise instead to wait for the bug fixes and the new documentation.

In the meantime, you might want to look at the RexxArpLib function library available on BIX or the Fred Fish disks. It provides many of the same capabilities, albeit in a different and much more restricted manner. It is, however, a good way to play around with ARexx and Intuition, and it can help you to decide whether RxTools is for you.

RxTools
TTR Development
1120 Gammon Lane
Madison, WI 53719
608/277-8071
$54.95
One megabyte required.

AmeegaView

Routine help for C.

By David T. McClellan

WRITING A PROGRAM that provides a highly interactive GUI interface can be a sticky proposition. Intuition is function-rich and correspondingly complicated to write for. ACDA Corp.'s AmeegaView (formerly AmigaView) is designed to make the environment easier to program in by making certain common operations simple and to provide consistent looks-and-feels for applications.

AmeegaView consists of a set of libraries, include files, demos, and on-disk documentation for Intuition, IFF, and graphics functions to call from SAS/C and MANX Aztec C programs. ACDA didn't address Modula-2; Benchmark and the other packages have their own simplified libraries. AmeegaView claims to run with Lattice C 5.02 and 5.04 (which became SAS/C as of 5.10), and Aztec C68/k 3.6a through 5.0b. The documentation comes on disk, designed to be printed out as you see fit.

The functions deal with familiar Intuition objects (screens, windows, menus, gadgets, and so on), as well as fonts, polylines, bitmaps, IFF images, and the like. The difference between Intuition and AmeegaView is the approach. With Intuition 1.3, to open a screen and a window and add some custom gadgets and menus, I have to fill in values on numerous large C structures and then call several Intuition functions to build the user interface. Using AmeegaView, I can cut out most of the data-structure initialization and some of the calls—the AmeegaView routines fill in the defaults for me, while allowing me to customize those I want. For example, here's an Intuition window-open as compared to the AmeegaView version:

```
/* Intuition version */

struct IntuitionBase *IntuitionBase;
struct Screen LDraw_scr;
struct NewWindow newwin;
struct Window *LDraw_win;

/* Open Intuition library, screen, allocate
bitmaps, then */

newwin.LeftEdge = 0;
newwin.TopEdge = 0;
newwin.Width = WIDTH;
newwin.Height = HEIGHT;
newwin.DetailPen = WHITE;
newwin.BlockPen = BLACK;
newwin.IDCMPFlags = CLOSEWINDOW |
   GADGETUP | MENUPICK |
   MOUSEBUTTONS;
newwin.Flags = WINDOWDEPTH | WINDOW
   CLOSE | SUPER_BITMAP |
   BORDERLESS | ACTIVATE;
newwin.FirstGadget = NULL;
newwin.CheckMark = NULL;
newwin.Title = (UBYTE *) "LittleDraw";
newwin.Screen = LDraw_scr;
newwin.BitMap = LDraw_bitm;
newwin.MinHeight = MINHEIGHT;
newwin.MinWidth = MINWIDTH;
newwin.MaxHeight = HEIGHT;
newwin.MaxWidth = WIDTH;
newwin.Type = CUSTOMSCREEN;
```

```
if ((LDraw_win = (struct Window *) Open-
    Window(&newwin)) ==NULL)
{
/* and so on */
}


/* AmeegaView version */

SCREEN LDraw_scr;
WINDOW LDraw_win;

/* . . . */

amigaview_open();  /* Init AmeegaView, open
Amiga libraries */

/* Open screen, allocate Bitmaps, similar to
intuition */
/*  then ...  */

LDraw_win = window_create (LDraw_scr,
   0, 0, WIDTH, HEIGHT,
   WINDOW_Input, MOUSEBUTTONS |
   CLOSEWINDOW |
      GADGETUP | MENUPICK,
      WINDOW_SpecialFeature, BORDER-
      LESS | ACTIVATE,

/* It figures out SuperBitMap */

   WINDOW_SystemGadget, WINDOWDEPTH
   | WINDOWCLOSE,
   WINDOW_Title, "Little Draw",
   WINDOW_BitMap, LDraw_bitm,
   WINDOW_MinWidth, MINWIDTH,
   WINDOW_MinHeight, MINHEIGHT,
   0);
/* end the argument list */

if (LDraw_win == NULL)
{
/* errors */
}
```

What AmeegaView does here is replace the process of filling in a long C struct and then calling an Intuition function with the simpler process of calling a corresponding AmeegaView function with a few fixed and a number of optional arguments. The window_create() function, for example, requires a SCREEN argument and the left/top/width/height components of a window. It fills in default values for the other arguments if you don't specify them. In the example above, I chose to fill in a few other slots with nondefault values, to tell AmeegaView which input events, special features, and system gadgets I wanted on the window. These optional arguments are listed in keyword-value pairs, as with WINDOW_MinWidth, MINWIDTH. These pairs can appear in any order in the list after the required arguments; the final 0 argument ends the list. I found this format easy to use—not much different from setting up a long printf. With AmeegaView filling in defaults for things I normally copied in from other, earlier programs, I cut down on source size. The expense, of course, was in executable size: The AmeegaView functions added 10–12K to a small program, more to a larger program using more of the library.

Windows and screens are a little easier with AmeegaView, yet not extremely so. But AmeegaView goes on to make menu-building, event- and gadget-handling, drawing, and IFF pictures easier, too. For example, the Intuition event-loop process is much simpler, from:

```
/* Intuition style */

struct IntuiMessage *msg;
ULONG msgclass;
USHORT msgcode;
struct Window *win;

for (;;)
{
   if ((msg = (struct IntuiMessage *)
      GetMsg(win->UserPort)) == NULL)
   {
      Wait(1 << win->UserPort->mp_SigBit);
      continue;
   }

   msgclass = msg->Class;
   msgcode = msg->Code;
   ReplyMsg (msg);
   switch (msgclass)
   {
   /* . . . */
   }
} /* end for */
```

to:

```
/* AmeegaView style */

EVENT anEvent;
WINDOW win;
/* . . . */
input_insert(win);  /* Let AmeegaView know to
watch for */
/*  events from win          */
for (;;)
{
   anEvent = input_read();
   if (anEvent == NULL)
      continue;
   switch (event_class(anEvent))
   {
   /* . . . */
   }
}
```

Again, the AmeegaView version is not trivial, but is much simpler than the Intuition version. And I can still get all the information out of the Message/Event that I want—the message class, gadget or menu ID, and so on.

AmeegaView provides simplified functions for the following kinds of Intuition objects: Screen, Window, Menu, Gadget, Image, Border, IText, Requester, and IntuiMessage. Other Amiga graphical objects supported include RastPort, Font, BitMap, Layer, multi-segment lines, and bitmap and color data associated with images. To use them, your program must include AmigaView/amigaview.h, link with the SAS/C or Aztec C library, and provide a C routine called clean_up() with no arguments, for exit processing. It also must call amigaview_open() before calling any other AmeegaView routines. After that, it can open screens and windows; build and modify menus, lists of gadgets, and requesters; and draw to its heart's content.

One of these objects is a very handy combination of several gadget features. A CHOICEGROUP uses a list of gadgets with mutual exclusion to provide similar functionality to RadioButtons for MicroSoft-Windows and the Macintosh. In such a group of gadgets, only one gadget can be "on" (selected) at a time; the user can click on an unselected gadget, turning it on and automatically turning off the previously selected one. These are handy for implementing "pick one of these" requestors.

I had some problems with the documentation. It was apparently written by a programmer and not very well organized. Most of the information I wanted was there, but it was full of distracting typos. Most of the routines are reasonably named; all WINDOW routines begin with window_, for example, but event routines began with event_, input_, and even multiinput() (although the docs said this one is obsolete). Capitalization is not consistent; for example, there is

Continued on p. 60

On Disk

An Introduction To Boopsi

The author of 2.0's object-oriented Intuition programming language helps get you started.

By Jim Mackraz

PROGRAMMING A NICE user interface under Intuition in the conventional way is more difficult and frustrating than it should be. If you are already familiar with Intuition, you'll find relief in OS 2.0's object-oriented programming support. The goal of the Basic Object-Oriented Programming System for Intuition (Boopsi) is to make things at once easier, more powerful, and more standardized. Boopsi is very helpful for writing 2.0 "new look" applications, especially in dealing dynamically with various font sizes and screen resolutions.

Before we jump into creating all sorts of new and interconnected custom gadget objects, we will start by using some of the ready-to-use classes of objects that are implemented in Intuition. Most of the important concepts of object-oriented programming will show themselves as we explore these simple examples, so if you're familiar with object-oriented programming, you will catch on quickly. (For a more technical discussion of Boopsi, see the 1990 Commodore-Amiga Developer Conference notes.)

ROUGH SPOTS IN INTUITION

First, a little stage setting is in order. We on the design team have learned a lot about Intuition's strengths and weaknesses in the last several years. Here are some fundamental areas of version 1.3 and earlier that we identified as needing attention:

• Intuition has proven to be at once too open and too restrictive. A button can be represented by any Intuition image, but nobody specifies or provides the best image to use. At the same time, the possibilities with Intuition images are rather limited. To see a standard look in applications, some standard and well-designed images have to be made easily available to applications.

• Constructing user interfaces from pieces is sometimes difficult. It's hard to encapsulate all the components of a scrolling list, for example, so that you can throw together a scrolling list and two command buttons in a requester and move on to the important part of the project: deciding how a scrolling list and a couple of buttons can bring benefit to the user.

• Intuition support is at a very low level; only the structures and constants are defined for the primitive image and gadget constructs. No routines are provided to dynamically allocate and initialize these nor to convert the data structure fields (such as VertPot and HorizBody) to meaningful quantities. Procedural wrappers around gadgets and images have to be provided by the applications programmer—each applications programmer.

• More problems crop up when you try to implement a refined look to all the gadgets and images. The primitive types provided are not sufficient, and you cannot mix text, line drawings (borders), and raster images in a linked drawing list. To create a button with a label centered in a filled-in beveled box, you must provide the text in one place, convert your algorithm for drawing beveled boxes into the proper-size image raster, and provide that raster in a separate place.

• Perhaps the biggest problem with Intuition is that it is not extensible. Every mechanical improvement in gadget functionality has to be implemented in ROM (or by complete replacement of the gadget mechanism). If programmers can be trusted to exercise good judgement and restraint with regards to standards, then providing programmers with means to specialize and extend the types of objects implemented in ROM allows us to reach a much greater level of refinement in how programs look and how they interact with the user.

BOOPSI COMES TO THE PARTY

On other computer platforms, object-oriented programming has proven itself suitable, almost ideal, for addressing these kinds of problems in graphical user interface systems. User-interface data structures are represented as *objects* with a uniform procedural interface, often called *sending a message to an object*. An object is not just a data structure, similar to a window, gadget, requester, or image, but also a handle on the executable subroutines that operate on the data and define the object's behavior. A *class* refers to support for objects of some specific type. If you wanted phone-number gadgets, you would implement the phone-number gadget class, and in your programs create from that class and use some phone-number gadget objects.

Boopsi is an attempt to bring some object-oriented programming to the Amiga, especially to Intuition programming. It shares principles with most other object-oriented programming systems, but I placed some unusual requirements on it, including:

• You should be able to use any programming language to write programs using Boopsi and predefined classes. If the programming language is sufficient to implement an Amiga interrupt handler, then you should also be able to implement new Boopsi classes.

• Boopsi must be small to fit in ROM, minimize its use of

RAM at runtime, and be reasonably fast on modest hardware. Remember, the core of the Amiga product line is the A500.

• The user-interface classes must be compatible and integrated with existing Intuition concepts. Boopsi has to fit into the rest of Intuition, not hang off the side. Support for the 2.0 new look was the project's highest priority.

• Applications and classes implemented for them have to be protected against future changes to the implementation of system classes. I could not expect every Amiga application using Boopsi classes to be recompiled because somebody at Commodore needed to add separate line-width and line-height parameters in the embossed-box image object data structure.

To abide by these rules, we had to accept that Boopsi was not going to be the be-all, end-all object-oriented application development environment. We decided if it satisfied these requirements and helped solve some of Intuition's problems, it would be worthwhile. It has already proven of great value in implementing the new look of 2.0 system gadgets, glyphs, and requesters.

CREATING BOOPSI OBJECTS

All of the benefits in the world are not going to matter if Boopsi is just too difficult to use. Let's take a look at Listing 1 and see how it stacks up. This code fragment creates an embossed box image, draws it into a window, and then discards it.

Listing 1:

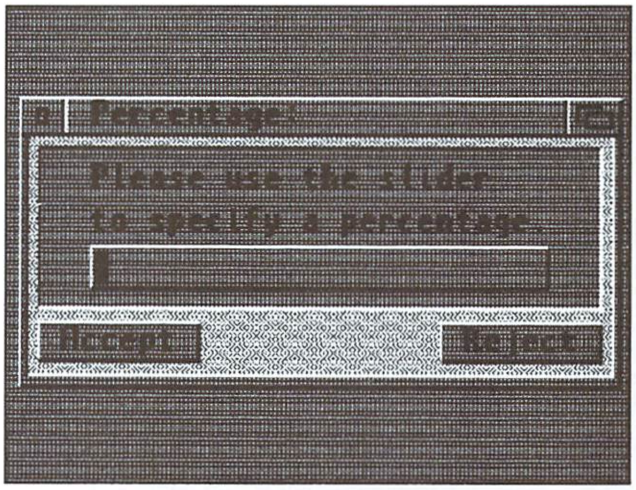
```
struct Image *boximage;

boximage = (struct Image *) NewObject( NULL, "frameiclass",
    IA_Width,  60,
    IA_Height, 40,
    TAG_END );

DrawImage( w->RPort, boximage, 20, 40 );

DisposeObject( boximage );
```

The first of the three program statements creates an image structure using the Boopsi function NewObject(). (Like almost all of the Boopsi functions, NewObject() is a function in intuition.library.) To create an object, you identify the class of object you want. Here, it is specified by the string frameiclass, which is an abbreviation of frame image class. This


Percentage

Please use the slider
to specify a percentage.

Figure 1: An enlarged view of a sample Boopsi requester.

class is *public*, in that it is available for any application's use and is referenced by its name. When you advance to implementing your own classes, you should start with private classes that only your application can use. Such private classes are identified by a pointer. Here, we pass NULL for the pointer because the class is public. The class we are using creates objects that behave like images. In traditional object-oriented programming fashion, frameiclass is a *subclass* of the *base* image class. It is said that frameiclass *inherits* behavior from imageclass.

The next several function arguments to NewObject() are *attribute tag-value pairs*. The symbols IA_Width and IA_Height are defined (in the include file intuition/imageclass.h) to constant values that identify (tag) the image width and height attributes. The IA_ prefix in the symbols stands for Image Attribute. Following each attribute tag is the attribute value, here some arbitrary numbers to specify the image width and height. You can specify as many attribute tag-item pairs as you want to NewObject(); you indicate the end of the list with the special tag value TAG_END.

Passing attribute lists like this in a function call with a variable number of arguments is actually just a device of the C language. The formal system interface behind NewObject() passes the address of an array of TagItem structures, which NewObject() just assembles on the stack. You can pass arrays, too, and even mix and match variable argument lists with fixed arrays. (See the documentation and include files for the new 2.0 utility.library for general information on using attribute tag lists, which pervade the new programmer interfaces throughout 2.0.)

You can use the function NewObject() to create *all* Boopsi objects, not only images. In general, an object is represented by an abstract pointer. Exactly what it points to is often private information. Ideally, all specification and access of object parameters would be done via attribute tags and values passed to functions NewObject(), SetAttrs(), GetAttrs(), and their equivalents. That would be obeying a true black-box interface.

As part of the charter of compatibility and making things fast and small, however, there are some exceptions. In particular, for image and gadget classes the function NewObject() returns a pointer to a familiar Intuition Image structure or Gadget structure, respectively.

You should severely limit your interpretations of the data fields in these transparent object structures, preferably just to ►

position and dimension fields for images and gadgets, and the NextImage field for images. When available, always use the formal attribute mechanisms for changing any data structure field; do not poke the objects.

USING AND DISPOSING OF IMAGE OBJECTS

Moving on to the second program statement in Listing 1, we see another compatibility trick. You can pass image and gadget objects to the Intuition functions that accept images and gadgets, respectively, as parameters. The function DrawImage() historically has rendered bitmap raster images using the Blitter. Here, however, Intuition detects that the image passed is a Boopsi image object and relies instead on the object's class implementation to perform the desired action: Draw an embossed box.

As you will see a little later, every action on objects, including draw, is accessed through a common interface function named DoMethod(), which is Boopsi's version of sending a message to an object. DrawImage() just calls DoMethod() for Boopsi images. (The second example calls DoMethod() directly, to illustrate the equivalence.)

The most interesting part of object-oriented programming is probably apparent already. We do not know how the embossed box is rendered or even how it is represented internally. There's a subroutine somewhere that does the right thing, and our only path to that subroutine is through the object. If we had an image class whose objects represented pictures of the phases of the moon, we might have even less idea how it really works, but we could call the exact same function, DrawImage(), to get the desired results. The point is that all we want to do is draw the thing, not be bothered with the details. Nor do we want to carefully call some DrawMoon() function when dealing with MoonPhase images.

"The power of object-oriented programming lies in polymorphism."

Applying the same operation to objects of a variety of classes, and getting the results appropriate for each type of objects, is called *polymorphism* in the object-oriented programming world. This is where the power lies.

If you've ever been deeply troubled by the fact that Intuition Image, Border, and IntuiText structures each have their own rendering function (DrawImage(), DrawBorder(), and PrintIText(), respectively), are not self-identifying, and cannot be intermingled in a linked list, then rejoice: You are already an object-oriented programmer, and Boopsi puts an end to this kind of nonsense once and for all.

The third and last statement in Listing 1 also demonstrates polymorphism. The function DisposeObject() deallocates every object returned from NewObject(), regardless of its class. This can help out certain schemes of resource tracking and cleanup; it is also convenient that you can pass a NULL pointer safely to DisposeObject() (and, more generally, to DoMethod()).

NEW-LOOK BUTTONS AND THEIR IMAGES

The 2.0 Amiga user-interface new look makes heavy use of embossed frames and silkscreened button glyphs. It is common that a frame must be calculated to enclose some centered image contents. Also, the colors used to draw system gadgets depend on the screen they appear in. To handle these and other new-look requirements, there is a bundle of useful information associated with each screen that the fancier images use to determine their dimensions and colors. The information is contained in the DrawInfo structure.

Listing 2 uses a screen's DrawInfo structure to render a "smart" image object.

Listing 2:

```
struct DrawInfo *drinfo;
struct impDraw  draw_params;

drinfo = GetScreenDrawInfo( window->WScreen );

/* a new generalization of DrawImage() */
DrawImageState( w->RPort, boximage, 20, 40, IDS_SELECTED,
   drinfo );

/* Same function without the wrapper, using stack-based
                    arguments.
                    */
                    offsetpair = (20 << 16) | (40);    /* make a ULONG */
                    DoMethod( boximage, IM_DRAW, w->RPort,
                       offsetpair, IDS_SELECTED, drinfo );

                    /* Same again, using DM() the packaged-
                    parameter version of DoMethod()
                    */
                    draw_params.MethodID = IM_DRAW;
                    /* identifies this struct */
                    draw_params.imp_RPort = w->RPort;
                    draw_params.imp_Offset.X = 20;
                    /* X and Y are WORDs   */
                    draw_params.imp_Offset.Y = 40;
                    draw_params.imp_State = IDS_NORMAL;
                    draw_params.imp_DrawInfo = drinfo;

DM( boximage, &draw_params );
```

Generalizing the existing function DrawImage(), a new function DrawImageState() has been introduced. This extends DrawImage() to provide a pointer to the useful DrawInfo structure and to specify which state you want your image to render, such as the selected state of a gadget button. Putting this knowledge in the images lets you fine-tune how gadgets should look in action, without needing to deal with more than one image per gadget.

Listing 2 also demonstrates two equivalent invocations of DrawImageState(). Like DrawImage(), DrawImageState() is also just a wrapper around the ubiquitous dispatcher DoMethod(). The example shows both a stack-based argument interface to DoMethod() and an alternative packaging of the parameters in a structure defined in intuition/imageclass.h. The parameters to DoMethod() depend on which operation or *method function* you are invoking. The *method ID* used for DrawImageState() and DrawImage() is IM_DRAW. This method ID dictates which operation the object will perform. The object's class inspects this ID and dispatches to the appropriate software.

FRAMES AND CONTENTS

Consider putting a text label inside an embossed box. How big do you make the box, and how do you center the text? Only when the intelligence behind the frame is coupled with the data in the DrawInfo structure can a perfect answer be determined. The font of the text is clearly important, but so is the "shape" and thickness of the frame. Ideally, the resolution of the screen's display mode should play a role in determining the thickness.

The accompanying example programs (in the Mackraz drawer on the companion disk) demonstrate a few different ways you can compose image frames with their contents to get the results you want. Centering text or pictures in embossed frames is only one part of the general layout problem, but it is enough to see the basic sequence of operations.

First, you have to rely on the intelligence of the framing object to tell you how big to make the frame for the contents to fit. Subsequently, you might want to add a little more space for aesthetics or to shrink or constrain the contents, if that size frame is too big to fit in the available screen space. Once you work out how big you want the frame to be, you can then rely on the frame-image object to tell you how best to center the contents within the chosen dimensions.

As far as your interactions with the frame are concerned, there are two steps: query the frame for its recommended dimensions and, after consideration or adjustment, specify to the frame which dimensions you want to use, and let it adjust (center) things appropriately. Both of these operations are invoked by a call to DoMethod() for the IM_FRAME method—this is the first method we have dicussed without a familiar wrapper function. Again, the parameters you pass to DoMethod() for IM_FRAME are defined by a structure in intuition/imageclass.h. One of the parameters determines whether you are inquiring about a recommended frame dimension or specifying the dimensions you decided on.

"The interface to data object values is best handled by using attribute tags."

A NEW-LOOK REQUESTER

Now we'll discuss the more interesting of the demo examples (boopsi_request.c), which creates a new-look-style requester with a proportional (slider) gadget in it. It is a "percentage requester" and prompts the user to use the slider to specify some percentage between 0 and 100, inclusive (see Figure 1). System requesters do not have sliders in them, so this is a fine example of extending the system by piecing together objects from predefined Boopsi classes.

In truth, the example really does not use a requester, but puts up the equivalent visuals in an Intuition window. For illustration, the program opens this window on the default public screen and uses the new Intuition functions LockPubScreen() and GetScreenData() to lay out the request window before opening it. It is still impossible to get a 100% reliable prediction of what the border dimensions of the window will be when it opens, but the program uses new 2.0 techniques to specify the inner dimensions and lets Intuition worry about the border thicknesses.

The system requesters, like our example, have a handful of button gadgets, and ours also has a proportional gadget to specify percentage. The buttons each have an embossed frame, but the program shares a single frame-image object between all buttons. The text labels in the buttons are also low-overhead in that no IntuiText structures are involved, just plain ASCII strings.

In addition to the frame image shared by the buttons, there are a few other image objects, including a special object to disguise the requester IntuiText as an image, a frame for the text and the slider, and an image object that renders a stipple fill pattern within the window interior. These images are all chained together and attached to an invisible surrogate gadget so that it is easy to position them together within the window borders and Intuition automatically updates all of the artwork whenever it refreshes the window gadgets.

BOOPSI SLIDERS

Boopsi_request.c uses Boopsi classes to create the requester gadgets as Boopsi objects. The button gadgets are quite specialized and are designed for the new look. They are particularly smart about placing a simple text label within the raised embossed border.

The proportional gadget is also created as a Boopsi object, and what would any repackaging of proportional gadgets be worth if we did not finally dispense with the incredibly difficult programmer interface to the "Pot" and "Body" values of proportional gadgets?

As we learned at the beginning, the interface to data object values is best handled by using attribute tags. Three new attribute tags are defined for propgadclass: PGA_Top, PGA_Visible, and PGA_Total. These names come from the common use of proportional gadgets as scroll bars. For that use, if you had 100 lines of text to display in a 25-line region, you would specify PGA_Total to be 100 and PGA_Visible to be 25. You would use the attribute PGA_Top both to specify and inquire the "top line" displayed. When displaying the first 25 lines of information, PGA_Top would be zero. At the other extreme, the top line of the last page of display would be 75. In a future article, I will show how you can arrange to receive IDCMP messages for slider dragging only when the value of PGA_Top changes, which is just what you want.

In our example at hand, we don't want a scroll bar, but a valuator that takes values from 0 to 100 and increments or decrements by one for each container click. We do this by specifying PGA_Total to be 101 (0 to 100 inclusive) and PGA_Visible to be 1.

WHEN DO WE START?

As with all of the new features of 2.0, it's a little difficult to know just when you can use Boopsi in new software products and have a sizable installed base of 2.0 operating systems out there to run them. While you wait, you might want to use Boopsi in non-product research projects as practice. When 2.0 ships to the mass market, I anticipate Boopsi will be one force behind an explosion of new and revised applications that share the attractive new look and have a new high level of user-interaction refinement. ■

Jim Mackraz is a consultant and software developer living in Palo Alto, CA. He was responsible for Intuition for five years. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (jmackraz).*

System Events and Event-Handling Routines

Find out who does what to whom.

By Eugene Mortimore

THE AMIGA'S COMPLEX multitasking software system can appear intimidating to even experienced programmers. One of the reasons for this apparent complexity is that many extremely vital routines were preprogrammed into the system, thereby saving you much work. Among these are the hidden hardware-interrupt routines that allow programs to interact with and respond to real-time outside events. While always hidden from your direct view, the globally present Exec-system internal routines maintain order in a system where outside real-time random events, intertask task-synchronized nonrandom events, and the machine-state transitions they bring about occur in split microseconds. To maintain order among all routines (system-predefined and programmer-defined) that need to share data, memory, CPU time, and other sometimes scarce system resources, the Amiga software system has a number of bookkeeping mechanisms.

As your programs become more complex, you may wonder: When is my task routine active? How did it become active? What is my task routine doing now? When are other task routines in the system active? Why do they become active when they do? How can I design my task to expedite its execution? What system-wide information can my task routine access? To answer these questions, this article will focus on the functional purpose and behavior of the most prevalent Amiga routine types. I will also try to make the descriptive language and the general interactions of Amiga processes, tasks, interrupts, and traps—and their associated routines—more precise, to help you place the tasks and routines you define in the appropriate total system context and understand the general logic of Amiga multitasking system operations.

Part of this article is concerned with software-interrupt events and their associated software-interrupt routines. Execution of these interrupt routines can be initiated by the Exec Cause function or through the use of software signals and their associated Exec message ports. Although I confined this article to the Exec Cause function mechanism for inducing software-interrupt events, it still illustrates the essential purpose and features of this type of interrupt routine without the added complexity required to discuss software signals.

THE EXECBASE STRUCTURE'S IMPORTANCE

Key to our discussion is the ExecBase structure, the primary bookkeeping mechanism for all concurrent tasks in the Exec software system. The Exec system-internal routines continuously maintain and update its parameters as task switching proceeds. This maintenance and updating—the continuous writing of some of its parameters—occurs on a very tight time scale. (While I will discuss only ExecBase structure parameters that relate to this article, you can find the complete definition in the execbase.h include file.)

Any task can look at (read) the ExecBase structure's parameters at any time to get a system-wide snapshot of the system's most vital characteristics. For example, the ExecBase structure TaskReady List substructure contains a system-wide list of all tasks currently ready to run in the system. Your task can examine the task names in that list (the list Node structure ln_Name parameter) to determine which tasks are currently ready to execute.

In a similar way, the TaskWait List substructure contains a system-wide list of all tasks currently waiting to run in the system. These tasks are usually waiting for a software signal. Your task can examine the task names in that list (again the list Node structure ln_Name parameter) to determine all tasks that are currently waiting to execute.

ExecBase's ThisTask parameter holds a pointer to the Task structure of the currently executing task. Also, its TaskExitCode parameter always contains a pointer to the exit code for the currently executing task. This exit-code routine could be designed specifically for the task—and referenced in a parent-task AddTask function call that created the task—or it could be the system's default task-exit code.

The TaskTrapCode parameter houses a pointer to the entry point of the current task's currently executing trap routine. This trap-code routine could be designed specifically for the task and have a trap number referenced in a task-related AllocTrap function call.

The IntVects[16] parameter always contains a group of 16 IntVector substructures defining the current system-wide hardware-interrupt routines. These IntVector structures define how all keyboard interrupts, vertical-blanking interrupts, and so on are currently treated.

Each IntVector structure contains a pointer to the code (iv_Code) and the data (iv_Data) of the currently effective hardware-interrupt routine assigned to a piece of Amiga hardware. These IntVector structures define the current hardware-interrupt routines that are used throughout the system. Upon system startup, most of these routines and their data are in ROM. When a specific task calls the Exec library SetIntVector and AddIntServer functions to replace or extend these hardware-interrupt routines, however, those newly-defined routines become effective throughout the system and will be used by all processes, devices, and tasks currently in the system. For this reason, make hardware-interrupt routine modifications and amendments with great care and keep an eye to their impact on other concurrent tasks in the system.

The five List substructures in the ExecBase structure's Soft- ►

NEW 1991 ELEASES!

SAVE $5! EACH

AMIGA WORLD
Presents
VIDEO TOASTER
DESKTOP VIDEO PRODUCTION SYSTEM FOR THE
AMIGA WORLD
Presents
HOT ROD YOUR AMIGA
Get the inside story on expanding your Amiga with powerful peripherals. Covers Amiga 500, 2000 and 3000 series computers.
AMIGA WORLD
THE AMIGA PRIMER
INCLUDES AMIGA 3000, SYSTEM 2.0 AND AMIGAVISION!!!

3 Exciting New Videos Packed With Inside Info And Hot Tips!

HURRY WHILE SUPPLIES LAST!

1-800-343-0728

CALL TOLL FREE or mail this coupon.

VIDEO TOASTER

The Video Toaster™ from NewTek is hailed as the world's first video computer card enabling broadcast-quality production on desktop! TheVIDEO TOASTER™ videotape is indispensable for Amiga owners considering the purchase of a Toaster™ or those curious about all the excitement over this "revolutionary breakthrough in technology."

VIDEO TOASTER™ provides in-depth, comprehensive information on the Toaster's™ wide array of features and amazing capabilities. Topics covered include installing the Toaster™ in the Amiga 2000; adding and testing other essential equipment; selecting source material; and manipulation of the many digital video effects, including flips, tumbles, mirrors, spins, splits and titles. This video also illustrates how to generate and then superimpose letters over pictures, how to produce three-dimensional animations and how to paint on video images.

See for yourself what the excitement is all about!

HOT ROD!

HOT ROD YOUR AMIGA provides authoritative advice on how to achieve maximum power with your machine, whether you own a series 500, 2000 or 3000 Amiga.

HOT ROD YOUR AMIGA teaches you how to expand memory internally and externally. It provides valuable, in-depth information on selecting and installing hard drives, memory boards and accelerators; back-up software and utilities; RAM and drive space differences; and other "hot-rodding" tips. It also covers high-end peripherals such as DCTV™ and the revolutionary Video Toaster™. Don't wait to soup up your Amiga!

PRIMER

THE AMIGA PRIMER video provides step-by-step instructions covering the many features of the Amiga. Whether you're a new owner or an experienced user, this easy-to-follow video will prove invaluable. Packed with almost 90 minutes of detailed information, THE AMIGA PRIMER teaches you in an entertaining format with vibrant graphics and upbeat music.

Gain the full benefits that the Amiga has to offer on all Amiga models, System 2.0 and AmigaVision®. It also covers the Amiga workbench, the CLI, peripherals and utilities. There's no easier way to master your Amiga!

YES! **I am eager to become an expert! Please send me the following videos:**

☐ Video Toaster™	~~$24.95~~	*$19.95*
☐ Hot Rod Your Amiga	~~$24.95~~	*$19.95*
☐ The Amiga Primer	~~$29.95~~	*$24.95*
☐ Animation Video, Vol. One	~~$29.95~~	*$14.95*
☐ Desktop Video, Vol. One	~~$29.95~~	*$24.95*
☐ Amiga Graphics, Vol. One	~~$29.95~~	*$24.95*
☐ The Musical Amiga	~~$29.95~~	*$24.95*
☐ Animation Video, Vol. Two	~~$29.95~~	*$19.95*

☐ Check/Money Order ☐ MasterCard ☐ VISA ☐ AmEx ☐ Discover

Make checks payable to TechMedia Video.

Please include $2.95 shipping & handling for one video, $5.00 for two or more.

Total Amt. $ ________

Acct. # ____________________ Exp. Date ________

Signature ____________________

Name ____________________

Address ____________________

City/State/Zip ____________________

TECHMEDIA VIDEO

TJ891

PO Box 802, 80 Elm Street, Peterborough, NH 03458 603-924-0100
An IDG Communications Company

Available in VHS only. Please allow 4-6 weeks for delivery. Foreign orders: add $7.50 for airmail delivery; $18 for two or more videos. Payment must be made in U.S. funds drawn on U.S. banks. TechMedia Video is the licensed North American distributor of AmigaWorld Videos. © 1989, 1990, 1991 by Razza Video USA. All Rights Reserved. Amiga is a registered trademark of Commodore Business Machines, Inc. Video Toaster is a trademark of NewTek, Inc. DCTV is a trademark of Digital Creations.

Ints[5] parameter define the software-interrupt routine queues of currently pending software-interrupt routines. The Exec system assures that software-interrupt routines always execute at a higher priority than tasks, but at a lower priority than hardware-interrupt routines. One use for software-interrupt routines is to modularize your tasks into micro tasks that execute on the side and do not require explicit task bookkeeping—no Task structure. Another is to allow any extensive-hardware-interrupt-related processing to be deferred until all currently pending hardware-interrupts are completed.

Finally, the ExecBase structure Quantum parameter is directly related to the time-slice round-robin time-quantum period allowed for each task to execute when round-robin multitasking is occurring. From now on, I will use the term Quantum to refer to the actual number of seconds between task switches. At present, the system sets Quantum to 16 in version 1.3 of the operating system, and to 4 in OS 2.0. While Quantum is a small time integral, it is a sufficiently long period for a large number of task (68000 assembly-language) instructions to execute before task switching reoccurs.

SYSTEM-CREATED TASKS VS. PROGRAMMER-DEFINED TASKS

As a programmer, you will spend most of your time programming and debugging your own programmer-defined tasks. As discussed above, in the most common circumstance, these program-process tasks will be automatically created by the system software whenever it creates an AmigaDOS process associated with your program. The Exec and AmigaDOS system software automatically work together to create a program-process task when your program is launched from the CLI or Workbench. At that time, the system software will, in essence, allocate and initialize a Process structure and its associated Task structure, automatically setting appropriate structure parameters to allow the system software to maintain the bookkeeping on your task.

While most of the time the system software handles task bookkeeping, occasionally you might create a task inside your program source code by allocating and initializing a Task structure and, in effect, simulating the system software's setup duties. In either case, it is helpful to know how your tasks fit into the total scheme of things: what your task-related programming responsibilities are, what the system does for you, and how the system behaves when your task executes and interacts with other tasks and other task routines concurrently in the system. The best approach to understanding these task-to-task interactions is to focus on the general sequence of system events and study the behavior of task and other routines that execute during that sequence. This way you will begin to fathom the functional nature of the various routine types and then be able to focus on the system's flow of control and relate it to your known programming requirements and the fixed constraints in the system.

To this end, let's take a look at the general flow of execution among programmer-defined-task and other task-related routines in our multitasking system. If you concentrate on the main principles highlighted here, you will better understand general Amiga programing principles and the specific sequence of task-related events, as tasks are continuously switched in and out of execution on a very-tight time scale. Your new understanding in turn will give you a better grasp of the context in which you program and in which your software executes. This will help you avoid competing with or duplicating the responsibilities and actions of the Exec or AmigaDOS system internal routines, letting you take full advantage of what the system can do for you. You can use these insights to better understand the interactions among your program-defined task routines and both system-predefined and programmer-defined task routines that are concurrently executing in the system.

THE GENERAL SEQUENCE OF TASK-EXECUTION EVENTS

We will focus on the interaction of programmer-defined task routines, predefined hardware-interrupt routines, programmer-defined hardware-interrupt routines, programmer-defined software-interrupt routines, programmer-defined software- and hardware- (68000 processor) trap routines, and the system internal routines. These categories cover all the types of routines you will need to consider. While there may be other mechanisms for creating and then initiating execution of these routines-—for, example, software signals—these are the only routine types understood by the Amiga multitasking system. Because Amiga multitasking is a complex subject, it is very useful to look at simplified situation that depicts the most important routine scheduling mechanisms.

Figure 1 illustrates two tasks and shows how the multitasking system works. Each could be a short graphics task, either a standalone task or an AmigaDOS process task, that draws figures in an Intuition window by calling Graphics library functions. Notice that I arbitrarily divided each task's code into four blocks. You could create these tasks either as explicit tasks within another program (spawn them as child tasks) or as separate program-process tasks whose simultaneous execution is somehow started from a CLI window or a Workbench icon.

The first block of code contains task initialization statements and is where you open libraries the task requires and allocate and initialize the dynamic structures the task uses. For a graphics task in an Intuition window, you would open the Graphics and Intuition libraries and allocate and initialize Screen, Window, and perhaps Rastport structures here.

The second and third blocks are for code that accomplishes operations specific to the task. For a graphics task, this would include calls to the Graphics library functions that draw graphics in your newly opened Intuition window.

The fourth block houses the code to clean up your task. Here you would close your opened libraries (Graphics and Intuition) and free any memory blocks you allocated in other parts—most probably the first block of your task code. If you modified the interrupt system, you would here restore the interrupts appropriately.

Figure 1 draws attention to the relationship between the Exec system code, the programmer-defined task code, hardware-interrupt code (system-predefined and programmer-defined), programmer-defined software-interrupt routine code, and any programmer-defined trap routine code assigned to the task. The Exec system code, while not explicitly shown in the figure, is the ever-present low-level code that manages everything else in the system; it is the system-software traffic cop. As our discussion unfolds, Figure 1 will also help clarify the functional distinction between hardware interrupts, software interrupts, and the two types of Motorola 68000 trap events in the Amiga system.

One of the keys to understanding the Amiga multitasking system is knowing the difference between synchronous and asynchronous events. This difference is easiest to understand in the context of two identical concurrent tasks. In general, if an event can occur at any time, it is an asynchronous event. If an event can occur only when a section of a task's code has been entered and a specific program instruction executed, it is a synchronous event. This distinction will become clearer as we discuss our two-task example in detail.

The two large rectangles represent the executable code of two separate tasks. The entry and exit points (first and last assembly-language instructions) for the tasks are at the top and bottom of the rectangles, respectively.

INTERRUPT ROUTINES AND EVENTS

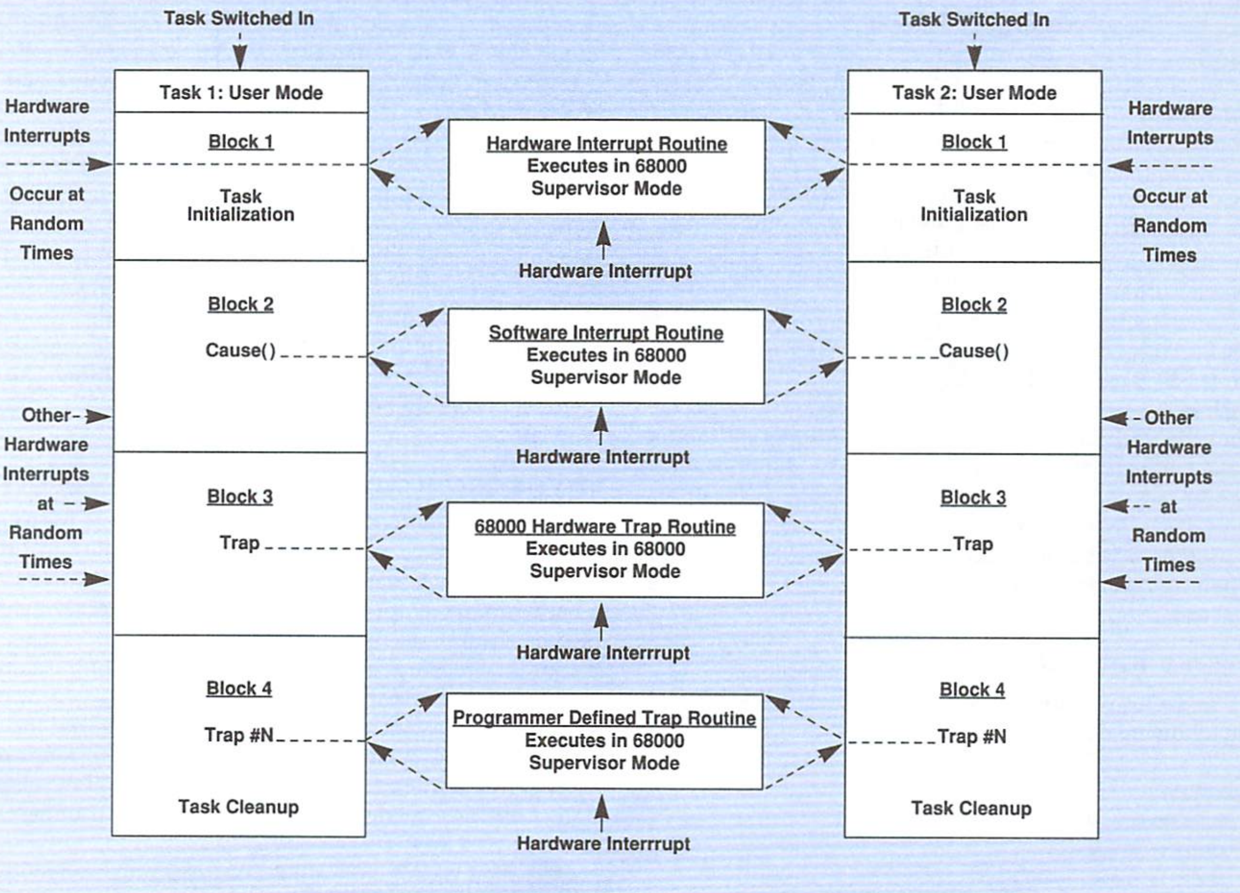
The inward-pointing arrows at the top of these two rectangles depict the action of the system's elapsed-time countdown routine that initiates task switching every Quantum seconds. The diagram does not explicitly show a separate rectangle for the system's task-switching routine, which is the routine that does the actual switch, changing task register contexts. In this discussion, it is considered to be one of the internal Exec system routines. You should understand, however, that the system's task-switching routine automatically gets the CPU every Quantum seconds as guaranteed by one of the Amiga's hardware-chip-countdown registers. Once the countdown event occurs, the system automatically initiates execution of the task-switching routine and task-switching occurs. You can classify the system task-switching event as either an asynchronous or a synchronous event. It is asynchronous because its occurrence is not determined by the task code. On the other hand, it is synchronous because it is regular and predictable in timing—it does not occur at random times.

The inward-pointing arrows on the outsides of the two task rectangles collectively represent all outside-world hardware-interrupt events that may occur randomly while the two task routines are executing. These arrows depict the source of most of the asynchronous events referred to above. Notice that these arrows point to random locations in the task code suggesting that they are induced by outside-world events that occur at random times during the execution of these tasks, such as the user pressing a key.

The four smaller rectangles in the middle of Figure 1 represent four types of additional routines. These routines may or may not be directly associated (assigned to the task by the programmer) with the two tasks. Notice that the entry and exit points (the first and last assembly-language instructions) for these four routines are at the top and bottom of their smaller rectangles. The inward-pointing arrows at the bottom of these four rectangles collectively depict all hardware-interrupt events that occur randomly while these routines are executing.

Thus, there is always the possibility that a new, second ►

Figure 1: Summary of Events and Event Handling Routines in the Amiga System.



Task Switched In
Task Switched In
Task 1: User Mode
Task 2: User Mode
Hardware Interrupts Occur at Random Times
Block 1
Task Initialization
Hardware Interrupt Routine
Executes in 68000 Supervisor Mode
Hardware Interrrupt
Block 2
Cause()
Software Interrupt Routine
Executes in 68000 Supervisor Mode
Other Hardware Interrupts at Random Times
Block 3
Trap
68000 Hardware Trap Routine
Executes in 68000 Supervisor Mode
Block 4
Trap #N
Programmer Defined Trap Routine
Executes in 68000 Supervisor Mode
Task Cleanup

(third, fourth, or more) randomly timed hardware-interrupt event will occur while the interrupt routine associated with a previous hardware-interrupt event is executing. The Exec system software must be able to cope with the nested-hardware-interrupt event situation. Hardware-interrupt events may also interrupt software-interrupt routines, 68000 hardware trap-routines, and programmer-defined trap routines. (This will become clearer later.)

FLOW OF CONTROL

The dotted lines between all rectangles in Figure 1 illustrate the flow of control in the system. When the 68000 program counter (address) points to the assembly-language instructions in each of the rectangles in Figure 1, that particular routine will be executing for a period determined by overall system logic and predefined system-scheduling considerations fixed by the Amiga's task-scheduling logic—a logic that is embedded in the hidden Exec scheduling routines. Specifically, in Figure 1, you can see that each task routine is deferred (interrupted) four times to allow a task-related side routine to execute. The dotted lines (between rectangles) and the associated terminal arrow heads show that when one of the tasks is interrupted, execution begins at the entry point of these "on-the-side" routines (the top of the rectangle) and ends at the exit point (the bottom of the same rectangle). Control then returns to the interrupted task at the same point that the task was interrupted, and its next instruction begins execution. Now the CPU program counter once again points to instructions inside the task routine itself.

For the sake of simplicity, assume that the two tasks have identical code, such as two copies of the same graphics program launched from one Shell. In that case, each task will inherit the task priority of the Shell window from which it was launched. For purposes of discussion, assume a 0 task priority; tasks will therefore execute in parallel alternately loosing and gaining the CPU. Under normal circumstances, every Motorola 68000 instruction that eventually executes in one task will also eventually execute in the other task before both truly finish execution. Except for premature task failure or user-task termination, this result is guaranteed by our two-identical task arrangement and by the way the Amiga multitasking system works.

No matter how it is brought into existence, each task is defined by its own Task structure and has its own distinct Motorola 68000 user-mode stack specified by its Task-structure stack parameters. When a program process task is executed from the Shell or Workbench, the system internal routines make these initializations. On the other hand, if you spawn a child task within an already executing parent task, you must do the initializations.

Assume that the variables task1 and task2 are defined as C pointers to the respective Task structures for our two-identical tasks:

```
struct Task *task1, *task2;
```

The Task structure, defined in the tasks.h include file, has the following definition:

```
struct Task {
   struct Node tc_Node;
   UBYTE tc_Flags;
   UBYTE tc_State;
   BYTE tc_IDNestCnt;
   BYTE tc_TDNestCnt;
   ULONG tc_SigAlloc;
   ULONG tc_SigWait;
   ULONG tc_SigRecvd;
   ULONG tc_SigExcept;
   UWORD tc_TrapAlloc;
   UWORD tc_TrapAble;
   APTR tc_ExceptData;
   APTR tc_ExceptCode;
   APTR tc_TrapData;
   APTR tc_TrapCode;
   APTR tc_SPReg;
   APTR tc_SPLower;
   APTR tc_SPUpper;
   VOID tc_Switch();
   VOID tc_Launch();
   struct list tc_MemEntry;
   APTR tc_UserData;
};
```

Several Task-structure parameters must be initialized before the task is added to the system with the AddTask function. Once the task is added, the system places it on the system's ExecBase structure TaskReady list. In our example case, one way or the other, each of these tasks is assigned 0 task priority by setting the Task structure Node substructure ln_Pri parameter to 0, which you do as follows:

```
task1->tc_Node.ln_Pri = 0; task2->tc_Node.ln_Pri = 0;
```

In addition, if the task uses its own trap routines, the tc_TrapCode and tc_TrapData parameters are initialized to point to the programmer-defined code and data for trap events and the tc_SPReg, tc_SPLower, and tc_SPUpper parameters are set to specify the task stack. In our simplified example, this is done identically for each task using matching C initialization statements (with differing Task structure pointers, of course).

Note that in this discussion, the Task-structure parameters, tc_ExceptCode and tc_ExceptData, used to define the code and data for software interrupts (exceptions) related to software signals, are not initialized because we are not considering software signals here. We are considering only the Exec Cause function mechanism of inducing software interrupts. The Cause function uses the Exec Interrupt structure to define the code and data to handle the software-interrupt event. Also, specifically note that the Task structure make no mention of code and data for interrupt routines. We will see why later.

THE DETAILED SEQUENCE OF EXECUTION EVENTS

Each task's four code blocks in Figure 1 (Block1 through Block4) can also be defined by the four types of events that occur during task execution:

- hardware-interrupt events and the execution of the associated hardware-interrupt routines
- software-interrupt events and the execution of the associated software-interrupt routines
- Motorola 68000 internal-hardware trap events and the execution of the associated hardware-trap-recovery routines
- programmer-defined software-trap events and the execution of the associated software-trap-processing routines.

Once the Task structure is allocated and initialized for each task, that task is added to the system with the AddTask function. When AddTask returns, both tasks are placed on the system's ExecBase structure task-ready list. Now the Exec system's hidden scheduler routines decide when the first task gets the CPU. If, apart from the ever-present system tasks, these were the only two tasks in the system, the system would operate as follows:

Phase One

Because, by assumption and arrangement, each of these tasks has equal zero priority, the multitasking system starts executing one of the tasks (probably the first one added with a AddTask function call), and Block1 of its code begins execution at its entry point. Quantum seconds later, Task2 begins execution. These two tasks each execute for Quantum seconds—first Task1, then Task2, then Task1, then Task2—alternately until, by assumption, a randomly-timed hardware-interrupt event occurs as shown.

Now assume Task1 was in the middle of executing a group of assembly-language instructions when the first hardware interrupt occurred. The interrupt is the outside world interacting with the task, asking for 68000 processor time to capture some information that is available only for a brief instant. This hardware interrupt could have come, for example, from someone typing at the keyboard or an AmigaDOS copy operation asking for information from the disk. The hardware-interrupt event could be completely unrelated to the task and therefore ignored, or it could be keyboard activity that supplies data for the task. In any case, as a result, execution of Task1 code ceases and the (short) hardware-interrupt routine associated with that particular hardware-interrupt event begins execution.

Just prior to passing control to the interrupt routine, the Exec-system routines automatically save the current register context (D0 through D7, A0 through A6 and the PC and SR registers) for Task1 on Task1's 68000 user-mode stack. This is the RAM stack controlled by the Task-structure stack-defining parameters as described above.

The Exec system-internal routines then put the 68000 processor into supervisor mode and the hardware-interrupt routine currently associated with that hardware-interrupt event begins execution. If it was a keyboard-interrupt event, the keyboard-interrupt routine places the new keystroke into the keyboard input buffer and very quickly returns control to the interrupted task. The Exec system routines then restore the previous task context (its saved registers) when the hardware-interrupt routine completes execution.

If this hardware-interrupt routine is part of a linked-interrupt-server routine chain, one or more of the routines in that chain execute, each in turn until the hardware-interrupt event is fully processed. This chained-routine arrangement is useful when you want more than one thing to happen with the hardware-interrupt event occurs. For example, the hardware-interrupt associated with a math-coprocessor hardware-interrupt event has a linked-interrupt-server routine chain mechanism. That is, at some point in the server-routine chain, determined purely by the preprogrammed internal decision-making process in the interrupt-server routines and the specifics of the hardware-interrupt event, server-chain routine execution ceases, the Exec system internal-scheduling routines redispatch Task1. The Exec system routines restore Task1's CPU register context from Task1's user-mode stack and Task1 resumes execution where it left off.

Here we assume no higher-priority task was added to the system task-ready list before the hardware-interrupt event occurred. This assumption virtually guarantees that Task1 will regain the CPU when the hardware-interrupt routine completes execution.

Phase Two

When multitasking and task-context switching resume Quantum seconds later, Task2 begins execution. The system alternately allows Task1 and Task2 to execute. The Exec-system internal routines automatically manage each task's stack, thereby saving and restoring each task's register context every Quantum seconds. Again, assume a randomly-timed external hardware-interrupt event—the same type of hardware-interrupt event as for Task1—occurs near the middle of execution of Task2's Block1 code. As with the first hardware interrupt, the timing of this hardware-interrupt event is random; someone hit a keyboard key. The system saves Task2's current register context and passes control to the same system-wide keyboard-interrupt routine used to process the first keyboard-interrupt event as for Task1. Then, very quickly, the hardware-interrupt routine returns control to Task2. Task2 begins executing again at the instruction following the point it was interrupted.

Phase Three

Quantum seconds later, multitasking again resumes; the system alternately allows Task1 and Task2 to execute every Quantum seconds. The system automatically manages each task's stack, saving and restoring each task's 68000 register context every Quantum seconds.

During Task1's Block2 code execution by task-design assumption, Task1 calls the Exec library Cause function. Cause tells Task1 to stop executing and transfer control to one of its software-interrupt routines. As before, Task1's register context is saved on its user-mode stack. (Note that the only argument in the Cause function call is a pointer to a Interrupt structure representing a particular software-interrupt routine.)

Now, just as with the previous hardware-interrupt routines, the system places the 68000 into supervisor mode to process the software-interrupt routine. Any function calls (such as shared-library function calls) the software-interrupt routine makes will use the 68000 supervisor-mode stack. Multitasking is suspended while the software-interrupt routine executes.

You must understand that the task prearranges all the actions of software interrupts that the Cause function induces by properly setting the parameters in an Interrupt structure related to the software interrupt. That is, the task initializes an Interrupt structure's is_Code and is_Data parameters in its Block1 code before calling the Cause function in Block2. Thus, the Exec Interrupt structure serves to define both the asynchronous, outside-world, randomly-timed hardware-interrupt event routines and the sychronous, task-timed, software-interrupt-event routines that the Cause function induces. The Interrupt structure provides a mechanism for a task to specify a code and data section to accomplish an arbitrary event-processing job, no matter what the source of that event. Also note that Cause-function-induced software interrupts are not defined by setting the Task structure tc_ExceptCode and tc_ExceptData parameters; these parameters are used for software signal-induced software-interrupt exception routines only.

Phase Four

Quantum seconds later, multitasking resumes. The instant Task2 regains control, it calls the Cause function and executes the same software-interrupt routine that Task1 previ-►

ously executed. The exact timing of this second software-interrupt event is guaranteed by the assumed symmetry of the situation; except for using different stacks and different Task structures, Task1 and Task2 are identical-synchronized tasks with one-for-one identical assembly-language instructions.

When Task2 turns over control to the software-interrupt routine, the Exec system routines automatically save Task2's register context on Task2's user-mode stack, and the appropriate software-interrupt routine executes in supervisor mode. Control quickly returns to Task2 routines when the software-interrupt routine executes an assembly-language RTS instruction.

Phase Five

Quantum seconds later, multitasking resumes. Task1 now regains the 68000 processor. The remainder of both tasks' Block2 code and part of their Block3 code executes. Once again, Task1 and Task2 alternately execute Quantum seconds apart. By assumption, at some point a Motorola 68000 hardware-trap event occurs in Task1. For example, the graphics task performs graphic-coordinate calculations that lead to an inadvertent zero divisor. When Task1 tries to divide by zero, the 68000 processor automatically senses this divide-overflow condition. One of the hardware trap-recovery routines defined by the system or programmer (the one designed to deal with zero-divide overflow) executes.

The trap routine should be designed to recover from the divide problem so that the Task1 can continue executing. For a interactive CAD graphics task, for example, this trap routine could alert the program user to the zero-divide condition and ask for the user to supply new program coordinates as input.

Just as with the hardware-interrupt routines, the system places the 68000 into supervisor mode to process the hardware trap-routine recovery code. Any function calls made in the trap routine use the 68000 supervisor-mode stack. Multitasking is suspended while the trap routine executes.

The system automatically saves Task1's register context on Task1's user-mode stack before the trap routine executes. The last 68000 assembly-language trap-routine instruction executed will be RTE (Return from Exception) and Task1 again quickly regains control. The RTE instruction places the 68000 CPU back into 68000 user mode.

Phase Six

Quantum seconds later, multitasking resumes; Task1 and Task2 alternate every Quantum seconds. Block3 of Task2 begins execution and, by assumption of symmetry, the same trap problem occurs: Task2 attempts to divide by zero. The 68000 senses this divide-overflow condition and execution of Task2 ceases while the divide-by-zero hardware-trap routine attempts to allow Task2 to recover. The trap routine finally executes a Motorola 68000 RTE instruction that returns execution to Task2. The RTE instruction again places the system back into 68000 user mode.

Phase Seven

Quantum seconds later, multitasking resumes; Task1 and Task2 alternate every Quantum seconds. Eventually, Task1 enters Block4 and executes a 68000 assembly-language TRAP #N instruction. This allows Task1 to execute an arbitrary programmer-defined trap routine to accomplish some job outside of the coding of Task1 itself—in effect, to perform a software-trap routine. The programmer has 16 software-trap routines available (16 software-trap numbers). 68000 software-trap routines are always initiated by the Motorola CPU 68000 TRAP #N instruction.

In this sense, the purpose of programmer-defined TRAP #N routines is very similar to software-interrupt routines except that their initialization mechanism is different. Software-interrupt routines are initiated by the Exec Cause function, by a software-exception signal sent through an Exec message port or by some other, more direct software-signal mechanism.

Just like a software-interrupt routine, the TRAP #N routine can be a large routine and is not time-critical as is a hardware-interrupt routine. Also like hardware- and software-interrupt routines, a trap routine executes in 68000 supervisor-mode, during which time multitasking is temporarily suspended. Therefore, note that if the trap routine is large, the trap routine itself should include a call to the Exec UserState function, to quickly switch back to 68000 user-mode from within the trap routine. This keeps the essential supervisor-mode processing short and allows multitasking to continue. These trap routines provide yet another way to modularize your code but, because they execute in 68000 supervisor-mode, are most often used in the Exec system code itself.

As before, the trap routine completely finishes execution and control returns to Task1. The last 68000 assembly-language trap routine instruction executed is RTE, and Task1 again quickly regains control. The RTE instruction places the system back into 68000 user-mode.

Phase Eight

Quantum seconds later, multitasking resumes and immediately, because of the guaranteed task symmetry, Task2 calls the same 68000 assembly-language TRAP #N routine, places the 68000 into supervisor-mode, temporarily suspending multitasking, and then completes execution and returns control to Task2. The last 68000 assembly-language trap routine instruction executed will be RTE, and Task2 again quickly regains control. The RTE instruction places the system back into 68000 user mode.

Task1 and Task2 continue executing Quantum seconds apart until their respective exit functions are called and control passes back to the system.

STILL IN THE QUEUE

By now you should have a grasp of the four types of routines that the Exec library functions can understand and manage. You can see that the system's CPU is constantly switching among task (and other) routines and between user and supervisor modes to accommodate all types of system events. Note that not all routines must be a direct part of an Amiga process or task. Instead, some routines can be called on the side as necessary. Keep this in mind as you explore ways to modularize your program code for maximum efficiency and reduce the Exec and AmigaDOS bookkeeping overhead.

If you study our discussion, you will begin to appreciate how your own program task fits into the total system, when that task accesses the CPU, and what considerations the Exec routines use to schedule tasks. To further this understanding, in the next issue I will expand upon this discussion's most important concepts and examine the implications and restrictions of 68000 supervisor- and user-mode execution and the proper use of task-related routines. ■

Eugene Mortimore is a developer and the author of Sybex's Amiga Programmer's Handbook, Volumes I and II. *Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

On Disk

PostScript Primer

With printer prices falling, the time is right to add precision graphics output capabilities to your programs.

By Jim Fiore

APPLICATIONS THAT DEPEND on high-quality hardcopy have no excuse for ignoring PostScript these days. PostScript dovetails quite nicely with the Amiga's graphic prowess, and it is becoming affordable as the prices for entry-level PostScript laser printers drop. I intend to wipe out your final excuse ("I don't know how to program in it.") by introducing you to the fundamentals of PostScript graphics and adding basic PostScript output support to your applications.

You do not need any sort of "PostScript compiler" to program with the language. PostScript laser printers contain PostScript interpreters. In effect, when you buy the printer, you are also buying a miniature language-development system, the interpreter. (See why PostScript printers cost more than ordinary laser printers?)

HELLO POSTSCRIPT

A page-description language, PostScript is device independent. In other words, the quality of the printed page is a function of the mechanics of the printer, not of the PostScript code. This is in contrast to the Amiga's graphics library, which is pretty much hardwired to the Amiga's chip set. A good portion of the PostScript language is built around the needs of transferring graphics and text to paper. The balance of the language consists of the elements you expect in any high-level language, such as facilities for looping, conditionals, procedures, definitions, and so on. Unlike such languages as C, PostScript is stack-oriented and employs postfix notation. In a statement you first place arguments onto a last-in-first-out stack, then give the procedure. If you had a procedure called smerzl that took two numeric arguments, a C language version would look like:

```
smerzl( 110, 23 );
```

The PostScript scheme yields a little different statement:

```
110 23 smerzl
```

Like C, PostScript is fairly freeform in nature. It lets you include blank lines and place several items on one line. The percent sign (%) denotes a comment. (Everything after % on a line is taken as a comment.)

When composing a page you should keep three things mind: the current page, the current path, and the clipping path. Empty when you start, the current page is where all drawing takes place. You draw on the page with opaque paint; there is no transparency. The current path refers to a collection of positions, arcs, lines, and so on, and is constructed using PostScript operators. This path can be very complex and can even cross over itself. The clipping path is the path that defines a printable area. For example, you might construct a clipping path in the shape of an ellipse. Only elements contained within this ellipse will be printed; anything that falls outside will be ignored. By default, the clipping path is equal to the dimensions of the page.

PostScript maps a piece of paper as an X-Y coordinate system with its origin in the lower-left corner. A positive X value moves you to the right, while a positive Y value moves you up the page. Note that this orientation is somewhat different than the Amiga's, which uses the upper-left as the origin, with positive Y values moving you down the page. The default unit is the point, which is defined in PostScript as one seventy-second of an inch ($^1/_{72}$). PostScript's operators will let you scale the units to your own desire, move the origin, and rotate the entire coordinate system. By doing so, you can work within your own ideal "user space." It is the job of the printer to map the positions onto its mechanics. This is how device independence is achieved.

Let's take a look at a simple program:

```
newpath
100 200 moveto
300 400 lineto
stroke
showpage
```

The newpath command clears the current path and declares that you are about to start building a new one. The second line places the values 100 and 200 on the stack and then calls moveto, setting the current position. The "pen" is now positioned 100 points to the right of the origin and 200 points above it. The moveto command is functionally similar to the graphics library's Move(). The program's third line draws a line from the current position to one 300 points to the right of and 400 points above the origin. Therefore, lineto is similar to Draw(). Note that no actual drawing has taken place yet. The stroke command initiates the process of drawing or painting onto the paper. Why is this extra step required? You can use the moveto/lineto commands to create a clipping path as well, and you may not want the clipping path to be outlined. The final command, showpage, indicates that all work on the page has been completed, and that the printer should create and eject it. While moveto and lineto are based on absolute positions, you can use rmoveto and rlineto to indicate relative positions. For example:

```
100 50 rmoveto
```

means move 100 points to the right and 50 points above the current position instead of to the right and above the origin. ►

As you can see, the moveto and lineto operators will allow you to create two-dimensional graphs with ease. Repeated calls to moveto and lineto will create an appropriate grid for the plot. Once this is completed, you can use a succession of linetos to connect the data to be plotted.

COPY TO PRINT

To see the results of the above program, the easiest method is to type the program into a text editor, save it, and send it to the printer by simply copying the file to the proper port. For example, if you connect via the parallel port and you call the program test.ps, the Shell command

```
COPY test.ps PAR:
```

will do the trick for you. (If you connect via the serial port, use SER: instead.) If you send an erroneous program to your printer, the device will usually just ignore it and do nothing. Depending on the exact error, it may eject a blank page. While this technique is wonderful for working out bugs in your code, it does not help you splice the code into your larger application.

As all your program needs to do is send strings of ASCII text from your program to the printer, the splicing methods are fairly straightforward. You could open up the appropriate DOS device (SER: or PAR:) and Write() to it. Generally, opening PRT: is inappropriate, because the printer device will perform character translations, and you want raw character streams. Another possibility is to use the Exec IO routines. *The Amiga ROM Kernal Manual: Libraries & Devices* (Addison-Wesley) includes snippets of code that are perfect for this, including a "plug-in" function called PrintRaw(). (Personally, I use the dissidents PrintSpool library as it also takes care of background printing. You can find it on Fred Fish disk #393.) Formatting the text strings is done in the normal fashion (C programmers can use sprintf(), or roll their own).

Shown below is a sample C program that uses the above PostScript code for a printer connected to the parallel port. For simplicity, it opens and writes to the DOS parallel device. The test.ps code is small enough to be placed into a single string. Larger applications will probably require the use of several strings and several calls to Write().

```
/* simple PostScript ps.c
For Manx 5.0,
c ps.c
ln ps.o -lcl
*/

#include "exec/types.h"
#include "functions.h"
#include "libraries/dos.h"

char ps_cmd[ ] = {"newpath 100 200 moveto 300 400 lineto stroke
   showpage\n"};

void main()
{
   BPTR file;
   long len;

   file = Open("PAR:", MODE_NEWFILE );
   if( file )
   {
      len = strlen( ps_cmd );
      if( Write( file, ps_cmd, len ) != len )
         puts("Write error\n");
      Close( file );
   }
   else
   puts("Open failure\n");
   }
```

GET FANCY

A single line is nice, but nothing special. With two new commands, you can modify the test.ps program to draw a box with a different line width.

```
newpath
100 200 moveto
100 400 lineto
300 400 lineto
300 200 lineto
closepath
10 setlinewidth
stroke
showpage
```

The setlinewidth command specifies the width of the line as 10 points. The closepath operator completes the box and produce a flush appearance. (Replacing closepath with 100 200 lineto will not give the same results. Because of the added thickness of the line, a small notch will remain at the start/finish point.)

You can create a filled box as well:

```
newpath
100 200 moveto
100 400 lineto
300 400 lineto
300 200 lineto
closepath
.6 setgray
fill
showpage
```

Also new to this example, the setgray operator allows you to create grayscales. For solid black use 0, and for white use 1.

Without some form of title, axis labels, and numeric scales, the graphs you build with the above commands will not be very useful. Enter PostScript's text-creation power. PostScript printers contain a set of standard typefaces (such as Times-Roman or Helvetica), but you can also instruct the printer to use typefaces that it does not hold internally. When you wish to print, you must first specify the typeface and size you want, that is, specify the current font. Here is a quick sample program:

```
/Helvetica findfont   % use the Helevetica typeface
12 scalefont   % make this 12 points high
setfont   % make this the current font
100 300 moveto
(Hello World) show   % print Hello World at location 100, 300
showpage
```

Note that you place the text to be printed within parentheses and use the show operator.

ALL TOGETHER

Now let's combine the graphic and text elements into one larger program that draws a filled gray box with a 10-point thick dark-gray border. Above and to the right of this box it prints blorg! at a 10-degree angle. (See Figure 1.)

```
newpath

% create a light-gray filled box
100 200 moveto
100 400 lineto
300 400 lineto
300 200 lineto
closepath
.7 setgray
fill

% outline the box with a dark-gray border
100 200 moveto
100 400 lineto
300 400 lineto
300 200 lineto
closepath
10 setlinewidth
.25 setgray
stroke

% make a 20-point Helvetica font.
/Helvetica findfont 20 scalefont setfont
0 setgray   % set the ink to black.
300 300 translate   % move the origin to the position 300, 300
10 rotate   % rotate coordinate system 10 degrees
50 150 moveto
(blorg!) show  % print the word blorg!
showpage
```

This example introduces two new concepts. First off, the translate operator lets you move the effective origin to any point you desire. After all, applications may require an origin somewhere other than in the lower-left corner. Also, a program might continually change the origin if multiple data plots or 3-D effects are used. The rotate operator moves the coordinate system a given number of degrees counterclockwise. The above example uses this to make the text run uphill.

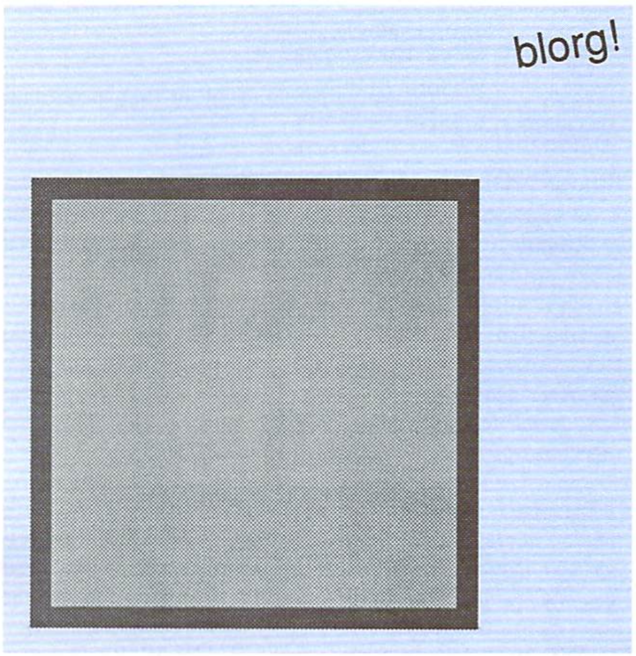


Figure 1. The setlinewidth and setgray commands in action.

Note that transposing the first two portions of the program will subtly alter the page. If the filled-box section is called second, it will obscure the inner portions of the surrounding border, because it is laid down using opaque "paint" (even when using grayscales, there is no transparency). Before moving to the next example, try experimenting with the clip operator. The basic idea is to define a clipping path using the appropriate newpath/moveto/lineto/closepath commands. When this path is complete, finish with the operator clip. This sets the clipping path to the one you just defined. Only areas within the clipping path will be rendered. For example:

```
% define a triangular clipping region
newpath
100 200 moveto
150 300 lineto
200 200 lineto
closepath
clip
```

Try placing this code chunk at the very beginning of the previous example and watch what happens (Figure 2). Note that the clip code affects only painting that comes after it.

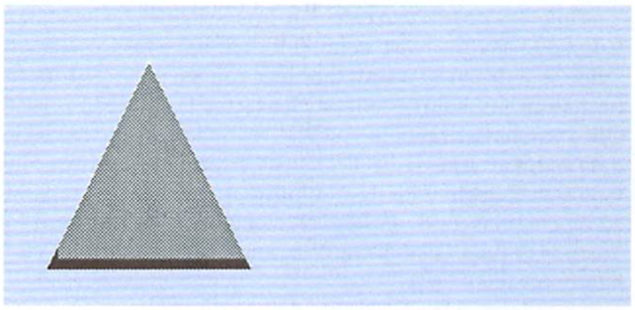


Figure 2. A triangular region clipped from Figure 1.

If you are tired of straight lines, a useful operator is arc. Its general form is:

```
x y radius begin end arc
```

The command

```
100 200 30 0 45 arc
```

produces an arc centered on coordinate 100, 200 with a radius of 30. The arc starts at 0 degrees from the horizontal and extends counterclockwise back to 45 degrees. (The arcn command produces a clockwise rotation.)

For a different perspective, consider the scale operator. Up to now, the default coordinate system of points ($^1/_{72}$ inch) was used. You can, however, scale the user space, increasing or decreasing the scales as desired. For example, if you would like to change the scale to something larger you could say:

```
100 200 scale
```

which increases the X axis by a factor of 100 and the Y axis by a factor of 200.

THE COMPLETE PICTURE

Our last stop on the PostScript tour is the transfer of images. Such operators as moveto, lineto, and arc are very useful for CAD-type programs, but they are not particularly efficient when dealing with bitmapped images. PostScript solves this problem via the image operator. With it, you can ►

transfer bitmapped images (black-and-white or grayscale). The image operator's general format is:

```
width height bits_per_sample transform_matrix procedure image
```

Width and height indicate the size of the image. Analogous to the "depth" of an Amiga screen, bits_per_sample can be a value of 1, 2, 4, or 8. For example, 4 means that each pixel is represented by a 4-bit value. A black-and-white picture has a bits_per_sample of 1. The transform_matrix is rather involved, but in essence, it allows you to map the image data onto the page in a variety of ways (for example, you could make a mirror version). For general-purpose image data where the origin is at the upper left, the transform matrix is [width 0 0 -height 0 height]. The procedure parameter is the data to be used. The image operator takes all of these parameters and maps the image data onto a "unit square." You can change the aspect ratio of this square with the scale operator.

A slightly more advanced use of the image command allows you to repeat the image (rather like a pattern) within the unit square. The data is organized as a series of samples, each being 1, 2, 4, or 8 bits in length. The value of a sample corresponds to the values associated with the setgray operator. That is, black is 0 and white is 1.

While it is possible to calculate image values by hand, it is tedious to say the least. A good tool to have for experimentation is a drawing utility that lets you save the picture information as C source code, such as Icon2C on Fred Fish #56. Once you have drawn the picture with the tool and saved its data file, you must edit it for PostScript consumption. Hex values in C begin with 0x and individual values are separated by commas. Both of these elements must be removed for PostScript data (search/replace and macros come in very handy for this).

The following program uses data derived from such a tool. The image is 32 by 20 and is black and white (one bitplane deep). The image will be drawn twice, so it is defined as an element called faceguy using the PostScript operator def.

```
newpath


% image is 32 wide by 20 high
/faceguy
<ff ff ff ff
f0 3f 00 ff
e7 9e 7e 7f
ff ff ff ff
f0 0f 00 ff
e7 e6 7e 7f
cf c6 fc 7f
e7 06 78 7f
f0 0d 00 ff
ff fc ff ff
ff fe 1f ff
e7 ff ce ff
87 1f 8e 1f
e7 c0 1e ff
f0 ff f8 ff
fc 00 03 ff
fd f9 df ff
fc f1 df ff
fe 1f 1f ff
ff c0 7f ff > def
100 200 translate
72 72 scale% 1 pixel = 1 inch
32 20 1 [32 0 0 -20 0 20] {faceguy} image

1 2 translate% move right 1 inch, up 2 inches
2 3 scale% twice as wide, three times as high
32 20 1 [32 0 0 -20 0 20] {faceguy} image

showpage
```

To see the picture, the coordinate system is scaled to a much larger size. Also, the translate operator brings the image up and to the right. The second time the program draws the image, a further translation keeps the two images on different parts of the paper. Note the second translate uses much smaller values than the first, because the user space has already been magnified by the 72 72 scale command. A second scale command shows how you can stretch or distort an image. In this case, the image will be taller and skinnier than normal. This all works fine for black-and-white images. Creating grayscale images is a bit more time consuming, however, because most of the available programmer's drawing tools save data bitplane by bitplane rather than producing values for each pixel.

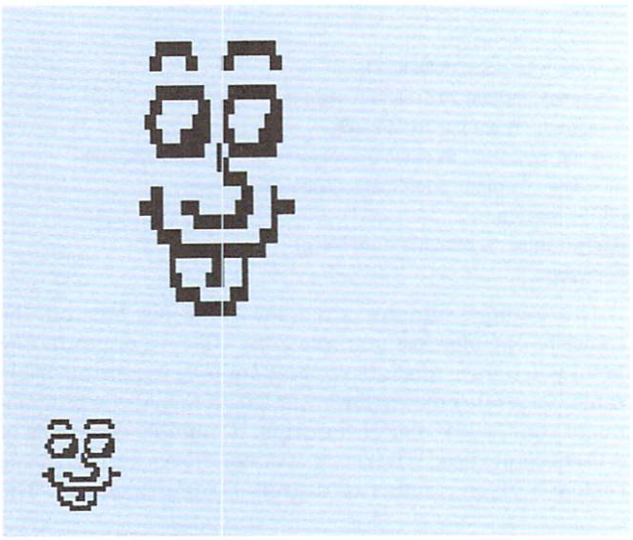


Figure 3. A repeated pattern element.

RELAX MODE

As you can see, PostScript programming is not particularly difficult if you have already mastered another high-level language. Once you test your code, splicing it into the Amiga application is easy. The commands discussed here should be more than enough to get you started, but if you would like to learn more about PostScript's power, I recommend *The PostScript Language Tutorial and Cookbook* and *The PostScript Language Reference Manual*, both by Adobe Systems Incorporated and published by Addison-Wesley. ■

Jim Fiore is a member of dissidents, which specializes in Amiga music and audio software. In his spare time he enjoys numerous hobbies and pursuits which have nothing whatsoever to do with silicon or C compilers. Also, he is presently putting the final touches on an electrical engineering text for West Publishing. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (jfiore).*

Become a part of the *AmigaWorld* Programming Team

We're looking for quality programs to support the growth of the *AmigaWorld* product line, and we need your help.

▼

GAMES
ANIMATION
3D
UTILITIES
CLIP ART
AMIGAVISION APPLICATIONS
OTHER STAND-ALONE APPLICATIONS

▲

We offer competitive payment and an opportunity for fame.
Send your submissions or contact us for guidelines:

Amiga Product Submissions
Mare-Anne Jarvela
(603) 924-0100
80 Elm Street
Peterborough, NH 03458

SAVE $5.00 NOW AND $5.00 LATER!

VIDEOS

Order now and save $5.00 off the retail price and get a $5.00 savings coupon good towards your next TechMedia Video purchase!

The Amiga Primer

NEW!

Anyone with an Amiga — whether a new owner or an experienced user — will find this easy-to-follow video invaluable. Applicable to all Amiga models, The Amiga Primer presents System 2.0 and AmigaVision™ the Amiga Workbench, the CLI, peripherals, and utilities in an entertaining format with vibrant graphics and upbeat music. There's no easier way to master your Amiga!

#TMAP 90 minutes ~~$29.95~~ $24.95

FREE!
Psygnosis Lemmings™ Game Demo Disk
with your order

NewTek's Video Toaster

NEW!

The Video Toaster™ from NewTek is hailed as the world's first video computer card enabling broadcast-quality production on desktop. Whether you are considering the purchase of a Toaster™ or are just curious about all the excitement, the Video Toaster™ video tape provides in-depth information such as:

- installing the Toaster™ in the Amiga 2000
- adding and testing other essential equipment
- selecting source material and processing speed
- manipulation of many digital video effects including flips, tumbles, mirrors, spins, splits and titles
- producing 3-D animations
- painting on video images

See for yourself what the amazing Toaster™ can do!

#TMVT 45 minutes ~~$24.95~~ $19.95

Hot Rod Your Amiga

NEW!

Learn how to get maximum power from your Amiga. Get valuable advice on how to expand memory internally and externally; select and install hard drives, memory boards and accelerators; back-up software and utilities; RAM and drive space differences; and other useful tips. Hot Rod Your Amiga also covers the revolutionary Video Toaster™ and other high-end peripherals such as DCTV. Soup up your Amiga now!

#TMHR 45 minutes ~~$24.95~~ $19.95

Animation Video Volume II

NEW!

Long-Awaited Volume II

In response to the clamor for another Amiga animation videotape, the AmigaWorld editors have created Animation Video Volume II. After sifting through hundreds of contest submissions and viewing countless hours of animation clips, the result is a showcase of scintillating animations, technical brilliance and imaginative plots. You'll be thoroughly entertained as you absorb new techniques and ideas. Whether you just brought your Amiga home from the store or have created your own animations before, you'll want to add Animation Volume II to your Amiga video collection!

#TMAV2 90 minutes ~~$24.95~~ $19.95

Desktop Video, Volume I

Whether for home or studio use, Desktop Video shows you how to utilize your Amiga in conjunction with a video camera to create professional-quality productions with live video and animation techniques. Learn how to select a genlock; build a desktop video system; choose and use a video camera; create scripts, storyboards and titles. Plus tips on editing, adding sound, special effects, and much more. A must for any video enthusiast!

#TMDV 84 minutes ~~$29.95~~ $24.95

Call Toll-free 1-800-343-0728

The AmigaWorld Catalog · PO Box 802 · 80 Elm Street · Peterborough, NH 03458

THE AMIGAWORLD CATALOG

The Musical Amiga

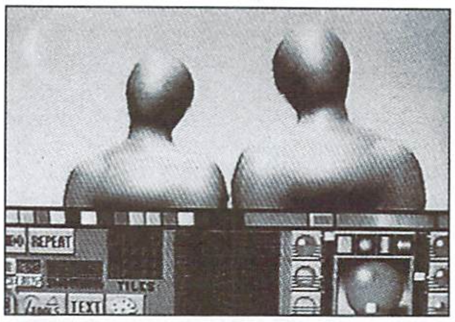
With detailed insight from some of the best-known Amiga musicians, The Musical Amiga teaches you how to create both simple tunes and complex musical compositions. Sessions on MIDI systems include sequencing, editing and synthesizing. Learn the techniques used in digital sound sampling, sound manipulation and creation of unique sound effects. Accompanied by an entirely Amiga-produced soundtrack, this dynamic video is sure to enlighten you with the remarkable musical potential of your Amiga!

#TMMA 60 minutes ~~$29.95~~ $24.95

Amiga Graphics

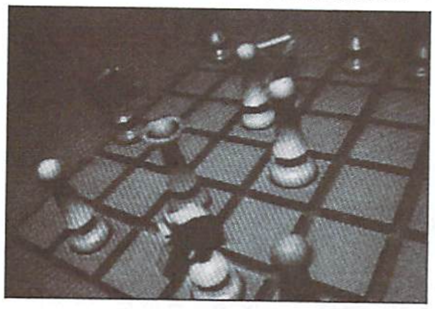
Amiga Graphics is all you need to explore the extraordinary graphics capabilities of your sophisticated Amiga. Learn the pros and cons of the most popular paint programs. Get three intensive sessions on fonts, clip art and digitizing, as well as in-depth coverage on the elements of design and style. Amiga Graphics also illustrates interlace functions, when Hold And Modify (HAM) should be used, and covers smearing, washing, tinting and perspective. An invaluable tool for any artist!

#TMAG 54 minutes ~~$29.95~~ $24.95



Amiga is a registered trademark of Commodore Business Machines, Inc.
Video Toaster is a registered trademark of NewTek, Inc.

Animation Video, Volume I



A best-selling video, containing commercially broadcast and award-winning animation art, this video will fascinate, entertain and inspire you as it demonstrates the amazing capabilities of Amiga animation. Each of the many animations is accompanied by unique music and sound effects, and is prefaced by the artist's name and a listing of all the animation products used. Experiment and explore the endless possibilities yourself!

#TMAV 48 minutes ~~$19.95~~ $14.95

AMIGA BOOKS

New Edition!

AmigaWorld Official AmigaDOS 2 Companion

by Bob Ryan

Harness the power of the new AmigaDOS 2! The Amiga Companion is back — in a revised and expanded edition for AmigaDOS 2. It's your expert guide to the new DOS, filled with hundreds of instructions, tips & techniques not found in any other book.

Covers:

- The Amiga OS, including Workbench, Shell & ARexx
- The Workbench GUI
- Workbench menus and tools
- The 13 Preference Editors
- The Extras 2 disk
- AmigaDOS — handling disks, files & devices
- Configuring AmigaDOS
- Manipulating files with Copy, Delete, MakeDir, Rename
- AmigaDOS command scripts and shortcuts
- The ARexx macro language and a fully annotated program to customize
- Plus over 100 screen shots, command references, a valuable glossary, and an error code summary for solving problems more easily!

#1-878058-09-6 416 pages **$24.95**

NEW!

AmigaWorld Official AmigaVision Handbook

by Louis Wallace

Express your creativity on the Amiga — with this complete guide to the new AmigaVision! The only authoritative guide to the hottest program for your Amiga. Written in an easy-to-follow style and heavily illustrated, with over 150 screen shots, this book provides a step-by-step primer for mastering AmigaVision.

Special Features:

- What exactly multimedia is, the art of authoring, the basic menus, common requestors, and program editing information
- Control commands covering: interrupts, database icons, wait icons, audiovisual icons, and module icons
- In-depth editors, tools, and programming information
- Valuable appendices filled with advice on the best hardware and software products for AmigaVision, and a special guide to version 1.7.

#1-878058-15-0 353 pages **$24.95**

Call Toll-free 1-800-343-0728

TJ891

The AmigaWorld Catalog • PO Box 802 • 80 Elm Street • Peterborough, NH 03458

Get Noticed

Convince the press to spread the word on your product for you.

By Brian Sarrazin

YOUR PRODUCT IS going to change the face of Amiga computing. If people knew about the technological wonder you created, they would beat a path to your door offering gobs of cash. The problem is they either do not know about your product or know about it but do not fully understand what it can do for them. So how do you get the word out?

Blanketing the market with ads and trade show demos will get you name recognition, but at a hefty price. If your budget is small, you are better off talking someone into doing the shouting for you for free. You need to convince the press to cover your product. Before you start dialing the phone, however, take these hard-learned lessons to heart.

SPEAK THE LANGUAGE

First, consider the perspective of your target audience and the context within which the message is delivered. You must understand how to speak the language of your listener to be understood. Communicating in a manner that is consistent with your audience's thinking style will also help.

With these thoughts in mind, you are ready to tailor your message to the experts in the press who can cover your product—the editors. Look at the mastheads of your favorite Amiga magazines. The editors listed are the people who will help you, if you help them first.

Editors pride themselves on objectively surveying the market and delivering important information to their readers. If editors feel that many people will benefit from your product, then it has a high level of newsworthiness. Make no mistake, a high level of newsworthiness is like having money in the bank. To convince them, you must tell the editors the benefits your product can deliver. To do this, study each magazine covering the Amiga market (and perhaps a few that cover the vertical market your product falls into), create and send the proper communication devices, and follow up appropriately.

Start by analyzing the magazines to garner an idea of the type of things the editors feel are important. It will also help to understand the editorial style. Read the feature articles in recent issues with an eye to grammar as well as to content. Look at the products reviewed. Scrutinize the helpful hints section for ideas on the type of issues the readers are wrestling with and, consequently, the concerns of the editors.

To further help you understand the editorial perspectives, request a media kit from each magazine. Inside the kit you'll find an editorial calendar outlining the magazine's planned emphasis for the coming months. Look for areas that your product may fall into and time your correspondence accordingly.

Learn the responsibilities and style of every editor on the masthead. Each will have a different reason for being interested in your product. The new products editor hopes to cover all that's new. The review editor hopes to coordinate and deliver objective summaries of key products' effectiveness. The features editor is interested in how products fit into key areas of functionality. Once you understand their interests and concerns, you are ready to contact the editors.

Editors prefer to be contacted first with printed matter, a press release and a cover letter. If you tailor your material to each one's style and perspective, you help them determine more quickly whether your product warrants coverage. Once decided, an editor may be interested in spending some time uncovering the details via the phone.

PUT IT ON PAPER

The press release is by far your most important tool to catch the editor's attention. As you only have about two seconds to do so, use your time wisely. Objectively report the facts and answer the five basic questions—who, what, when, where, and why—in as few words as possible. The pyramid approach is best: First supply the bare facts, then present the facts with some elaboration, and finally give the facts with significant elaboration. The most difficult part will be keeping your statements objective. You know how tremendous your product is, but you must contain your enthusiasm. There is no room for subjective opinion in a press release.

Structure the page for a quick read. The first line should tell the editor who to contact for more information. Underneath the name, list the phone number only. Next, you should indicate when the information may be released to the public, usually "FOR IMMEDIATE RELEASE." After this comes the headline, which must get attention. Indicate the most noteworthy aspect of your product here with a minimum of words. Adjective count: zero. Write that your company announces (or begins shipping) Product A and then state your most important reason for creating it. Double-space the remaining copy and leave generous margins to give the editor plenty of room to make notes.

Write the press release to mirror the target magazine's style, so the editor merely has to cut and polish it to suit the magazine's new products or news column. This helps the editor and it helps you communicate more directly to potential customers who read the magazine. Your goal is to write a press release that the editor does not have to change at all. Keep in mind, however, editors often will not print what they cannot confirm. If your product is not in the editor's hands, do not be surprised if your confident "WonderDrive is three times faster than existing drives" is changed to

"WonderDrive claims to be three times faster than existing drives."

Keep the first paragraph compact. Start by stating where the newsworthy event took place: where you began shipping your product or made an announcement. Was it at company headquarters, a trade show, a special press conference? Next indicate the date of the event. You have now covered the who, what, when, and where. Next, expand by more fully describing the what. For example:

New York City, New York, September 1, 1990 — Big Time Publications today announced a new magazine to be called *Lawn Care Journal*. Starting with its first issue in early 1991, *Lawn Care Journal* will cover items of interest to professional groundskeepers and will be edited by Jean Hobbs.

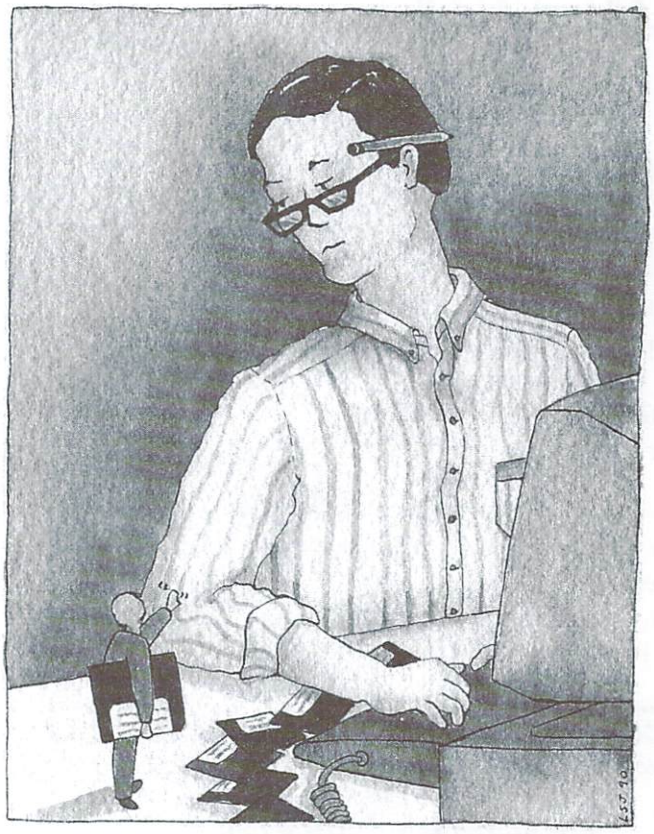
In the second paragraph introduce the why. Why will the Amiga community benefit from your product? Because the answers lean toward the subjective inherently, a quote does the job best. For example, just fill in the blanks of the sample template below:

"We feel there is a tremendous need for ___________ based on what we hear during on-line roundtable discussions. For this reason, we have created the first ___________ for the Amiga computer. It will revolutionize the way people ___________," said ___________, president of ___________.

The remaining paragraphs should go into increasingly greater detail and should include another quote. Don't get carried away; a press release should rarely be greater than 300 words. The above two examples total nearly 100 words already.

The cover letter lets you get into the why in detail and is essentially your opportunity to persuade the editor that your product is very newsworthy. Tailor each letter to the needs of the editor receiving it. Do not print 20 cover letters and 20 press releases and send them out to "Editor" or "Dear Sir." This is sure to alienate as it indicates a low level of concern for the editors' individual interests. A half-hearted effort will get you the results it deserves.

Remember: The less hype the better. Start your letter with a brief, 25-word statement of why you created your product. You should be able to use the first quote from your press release. If you can't, rethink your press release. In the second paragraph weave key features into your discussion of how you solve the users' needs. This shows that you put serious thought into the features you included, and maybe the ones you left out as well.



If your product is shipping, send it with the press release and letter. This gives your statements more weight, as the editors can see your product is real. If the product is still in development, send review copies as soon as it is ready. Do not fall prey to the "If they want to look at it, they can buy it" philosophy. If you do, your product will be relegated to obscurity. Remember, the press is helping you. They communicate with a very large group of Amiga users for you. Give away review copies liberally. In the case of hardware, simply loaning a unit is acceptable. You are still better off, however, sending a copy for keeps. That way the editors will have your product on hand if they schedule a related article a few months after it was released. Don't send a crippled version. The people of the press are professionals. You insult their integrity by sending a disabled version.

The last item to slip into the press package is a picture of the product, the packaging, or a sample screen. Look through the magazine to see which is preferred. The picture should be in slide format and on a disk as an IFF file.

FEEL FREE TO CALL

Follow-up phone calls are your next step. Again, vary the timing and purpose based on each editor's needs.

New products editors move at a quick pace. They have too much to report and not nearly enough time. Show you understand their time constraints. Make sure the significance of your product is evident from reading the press release alone. Limit your follow-up call to introducing yourself, ensuring that the package was received, and inviting the editor to call you back if needed. The timing here is critical. The sooner you ►

ILLUSTRATED BY LAURA STEARNS JOHNSON

call, the better, but do not call until you are sure your package is on the editor's desk. Sending the material via overnight or second-day express is very effective.

Review editors look for in-depth information. Do not attempt to provide it all in the press release, but make sure to explain why your product advances the state of Amiga computing, not merely why you made it. Follow-up is very important. Remember that a fair review could be the best thing that ever happened to your product. Your objective should be to clarify the reasoning behind your product and to determine where to send review copies. You will probably be asked for two copies. One will be for the editorial office and one for the person writing the review. (Do not ask for the reviewer's name; this is not professional.) Allow a couple of weeks to pass before calling. Realize that there are too many products vying for too little space and scheduling is a key aspect of the review editor's job. A two-week delay between receipt of the package and your call reflects your understanding of this. When asked for a review copy, send it express and call to ensure it arrived on schedule. Always notify the review editor of any updates that might affect a potential review or one in progress.

Features editors look for the most comprehensive information and work the furthest in advance. Consult your editorial schedule and look at the feature articles or issue themes coming up. Find one that most closely fits your product and explicitly mention the relationship in your cover letter. Indicate precisely how your product fits into the area to be discussed. (Keep the magazine's lead time in mind, however; editors often are working on issues four or more months ahead of the cover date.) A couple weeks after you send your package, call to ensure it arrived and offer to send anything else the editor might need for upcoming articles. Watch the editorial calendar and call again about the time the features editor plans the issue for which your product is appropriate. As for the review editor, alert the features editor to updates to the product that might affect coverage.

DON'T TAKE IT PERSONALLY

If you encounter editorial resistance to covering your product, do not become defensive or hostile. These people know more about the needs of their readership than you do. Your job is to make the benefits of your product clear. If you have done so and the editor decides that it is not worth covering, you have just received a very valuable lesson. Find out why they feel as they do. Perhaps they could direct you to a more appropriate magazine to cover your product.

If the editors decide to mention your product, find out exactly when the issue it is appearing in will be on the shelves. Time your discussions with other key community players (such as software resellers or publishers) with the release of the magazine; it will add credibility to your product.

The importance of communication cannot be overemphasized. No one will be interested in your product if they do not understand what it can do for them. If you think in terms of helping the experts who can help you, you will earn the press coverage your product deserves. The result will be throngs of Amiga owners beating a path to your door offering gobs of cash. ■

Brian Sarrazin is the Western Regional Manager for RGB Computer & Video. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

Start-up Messages

Pay attention, maybe your Amiga is trying to tell you something.

By David W. Martin

AS YOU NO doubt have noticed, when you boot under OS 1.3 your screen flashes a series of colors. Aside from simply loading the Workbench, your Amiga goes through a series of initialization and test procedures when you turn the power on or reboot. Not simply reassurance that "something is happening," the colors provide information on the results of these tests. Watch the flashes: If you learn how to interpret the colors, they can provide early warning signs to system failures.

Each time you boot, your system conducts the verification and initialization steps below:

1. Clear hardware data registers (custom chips, and so on) of old data.
2. Disable interrupts and DMA during testing procedure.
3. Clear the screen display.
4. Test the hardware and check that the CPU is working.
5. Change screen color to show the results.
6. Perform a checksum test on ROM.
7. Change screen color to show results.
8. Begin the system startup begins.
9. Test the $C0000 RAM and place SYSBASE information there.
10. Conduct a test on chip RAM.
11. Change screen color to show results.
12. Test to check if software is coming without errors.
13. Change screen color to show results.
14. Prepare chip RAM for data.
15. Link Amiga libraries.
16. Test for and additional memory expansion and link it if found.
17. Restore interrupts and DMA.
18. Begin execution of a default start-up task.
19. Test for usage of a 68010, 68020, 68030, or 68881/2.
20. Test for exception (or processor) error.
21. If an exception occurs, reset the system.

A healthy system flashes the following sequence on screen as the tests proceed:

Dark gray
Light gray
White

The first color indicates that the CPU test was passed and the 680x0 is up and running fine. The second reassures you that the software is coming in without problems. Finally, the white flash signals that the system passed all initialization tests.

Upon encountering trouble, the system changes the screen to one of the danger sign colors and halts all activity. A red screen indicates a ROM checksum error occurred. Green denotes an error in chip RAM, while blue tells you one of the custom chips has a problem. A yellow screen reports that the 680x0 found an error before the system's error-trapping software (guru) was started. While you probably will never receive an error message, the most common is the green screen. Many early Amiga 500s had loose Agnus chips, which caused chip RAM errors. Gently, but firmly, pushing in the chip usually solved the problem.

A SYSTEM ALL ITS OWN

While the CPU is the most obvious about its activities, your keyboard undertakes its own start-up sequence when you power up or reboot. A self-contained system, an Amiga keyboard houses a CPU, 2K of ROM, 64 bytes of RAM, four eight-bit I/O ports, and a timer, all of which are monitored by error checking and initialization software. When you boot your machine, the keyboard routines:

1. Check ROM with a ROM checksum test.
2. Check RAM with a RAM test.
3. Check the timer to ensure it is functioning.
4. Synchronize the keyboard with the computer.

To alert you to any errors it encounters, the keyboard blinks the light on the Caps Lock key. The number of blinks identifies the error. The blink code is:

One	Keyboard ROM checksum failed.
Two	Keyboard RAM test failed.
Three	Keyboard watchdog timer test failed.
Four	A short exists between two row lines or one of the seven special keys (currently not implemented).

Chances are you will never encounter an error flash or blink. If your system does start acting strangely, however, pay close attention at startup for these warning signs. While they will not tell you the whole problem, they will point you and your repair technician in the right direction, saving him time and you money. ■

David W. Martin is the author for the 1581 DOS Reference Guide *and a frequent contributer to Amiga publications. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on PeopleLink (davidm), CompuServe (72510,3232), or Usenet (davidm@hackercorp.com).*

From p. 37
event_MouseX() but event_menu-subitem(). There was no index. The documentation appears to have been set up for on-line browsing rather than paging through a printed copy; the files are organized one per routine in object-grouped subdirectories, which is only good if you have hard-disk space to spare and a multiwindow editor.

I tried the package with SAS/C 5.10 and Aztec C 5.0a, by adapting some existing code. The sources with amigaview.h included compiled acceptably, the objects linked, and the executables ran. I didn't notice any obnoxious slowing, and the gadgets and menus worked just fine. The add-on size of the library was not out of line with what I expected.

Using AmeegaView routines reduced typing time, and to some extent simplified the edit-compile-test cycle. Don't expect a no-brain solution, however; you can't use the routines without knowing what Intuition does and what its data structures specify. New programmers will still have to learn window and screen setup and Intuition/AmigaDOS message processing. AmeegaView merely makes it easier to deal with the nitty-gritty of typing in and compiling all those specifications. It will make your programs a little more readable and a little easier to test. It won't make Intuition any less complex.

AmeegaView is a good package, but a real manual and better naming conventions would make it a better one.

AmeegaView
ACDA Corp.
220 Belle Meade Ave.
Setauket, NY 11733
516/985-1700
$79.95
No special requirements.

MINIX 1.5

A little Unix for the Amiga.

By Stephen R. Pietrowicz

A UNIX-COMPATIBLE operating system, MINIX (which stands for Mini-Unix) was written by Andrew Tanenbaum, a professor at Vrije Universiteit in Amsterdam, The Netherlands, as a teaching tool for his classes. The Amiga version of MINIX offers a relatively inexpensive way to learn how an operating system, and Unix in particular, works.

Amiga MINIX is actually a port of a port. The original IBM PC version of MINIX was ported to the Atari ST, and that version was ported to the Amiga by Raymond Michels and Steven Reiz. Interestingly enough, under MINIX, the Amiga and Atari ST can exchange and run binaries without modification.

You'll find most of what you'd expect on a regular Unix Version 7 system in MINIX and a good working knowledge of Unix and it's utilities is essential. The package includes nearly all of the standard Unix utilities (over 175), a Bourne shell work-alike, a K&R-compatible C compiler, five editors, and over 200 library procedures. Additionally, because the utilities and the operating system code were written from scratch, MINIX can be sold without having to deal with licensing restrictions for the source code. In addition, the package includes complete source for everything except the C compiler and one of the text editors.

MINIX will run on a one-megabyte Amiga 500 or an A2000, and it will also recognize expansion RAM and extra floppy drives. MINIX does not, however, support the 68020, 68030, or, in the release version, hard drives. However, there is a beta version of MINIX that supports the A590 controller for the Amiga 500. If you have access to Internet, you can download the beta version that supports the A590 from star.cs.vu.nl at address 192.31.231.42. (It's probably also available at the sources mentioned later in this review.) Even if you don't have an A590 controller, I encourage you to find the new binary of MINIX. It seems to make the floppy disks work much more reliably.

SETUP

MINIX comes on nine disks—one (your boot disk) in AmigaDOS format and the rest in IBM format. You can use a commercial or PD IBM disk-copying program, or use the MINIX "diskcopy" command after booting MINIX to make backups. If you use the latter method, you must first format the backup disks using the transfer utility before booting MINIX. Format a few extra disks, because the Amiga version of MINIX doesn't have a formatting utility. If one of the disks is bad, you'll have to reboot AmigaDOS to reformat another. I usually keep a few extra disks formatted to avoid having to reboot. MINIX seems to be very particular about its floppy disks, and it rejected many that I tried.

When the system boots, it loads the root partition completely into RAM. The other partitions, such as /usr, can then be mounted on one of the external floppy drives. You can also increase the root partition's size to allow more binaries to be loaded into RAM and significantly increase the speed at which programs will load and run.

UTILITIES

MINIX comes with many utilities (disk utilities, text formatting utilities, communication programs, and more), and most work well. For example, I used kermit to test the serial device. I was able to dial up a local Unix system and transfer files back to my own system without any problems.

A few other utilities, such as the vi editor, didn't perform as well on the first try. The /usr disk as distributed was not set up correctly; a directory that vi uses to store its temporary files was missing. After I created the directory, all went smoothly.

I also had a problem trying to extract some of the source-file archives while attempting to recompile the entire operating system. All of the source comes in compressed archives. Decompressing the archives seemed to work correctly, but when I tried to extract the source from the resulting files, I didn't get very far before receiving an error message. I tried at least five times with the file-system source archive, and each time I received a different error message. Because I was unable to overcome that problem, I was unable to recompile the OS as I had hoped. I should also note that not all archives exhibited this problem. I was able to extract some of the source from different archives.

The need for hard-disk support was very evident when I started using the compiler. It expects to run out of the /usr directory, and because the compiler and the utilities on the /usr disk won't all fit on one disk, I had to rearrange things a bit to get a working compiler disk. Luckily, most of the compiler disk is already put together, so it isn't that difficult. Compiling programs on floppies is pretty slow, though. A simple four-line "hello, world" program took 1 minute and 45 seconds to compile to executable.

THE MANUAL

Amiga MINIX users will find the manual a bit disappointing, because it contains very little Amiga-specific information. Amiga owners are almost always referred back to the instructions for the Atari ST. To be fair, most of the information supplied will work for the Amiga, but there are a few things in the manual that are just plain wrong. For example, the information on creating a larger root partition is incorrect for the Amiga. When I followed the Atari ST instructions (as advised), I was unable to get MINIX to complete loading the RAM disk and then ask for the /usr partition. Everything hung up after loading the RAM disk. After a little experimenting, I found that using a different device name for the floppy disk I was creating worked. The manual recommends running a system verification test when you boot MINIX for the first time. I tried to run the test, but it wasn't even included on the disk!

The rest of the manual looked fine. It contains a complete explanation for each of the commands, along with examples. There are also extended pages for more complicated utilities such as kermit and the text editors. Plus, the manual has a brief, two-page explanation of the 49 system calls that MINIX supports and the source listing for most of the OS.

SUPPORT

You can get your MINIX questions answered from a variety of sources. The best is the comp.os.minix newsgroup on USENET. You can speak directly with Andrew Tanenbaum, as well as Steven Reiz. The group is in the process of reorganization, so by the time you read this, there will be a separate group for binary and source distributions. If you don't have access to USENET, you can participate in MINIX discussions on the Mars Hotel BBS, 301/277-9408. If you live outside the US, call NLMUG-ONLINE in Holland at (02522) 18363.

For more in-depth information about MINIX, you can pick up the book *Operating Systems: Design and Implementation* by Andrew Tanenbaum, from Prentice-Hall. Although it refers to an earlier version of MINIX, most of the concepts still apply. If you plan to do any serious work with MINIX, I recommend you get the book.

FINAL THOUGHTS

I have mixed emotions about MINIX. The package has a lot of promise, but it falls short in some areas. For example, such features as networking and hard-drive support that are supported on other platforms are not available for the Amiga. I hope this is rectified in the next version. If you don't have access to any of the support sites listed above, you'll have a hard time with MINIX and I suggest waiting for the next release. ■

MINIX 1.5
Microservice Customer Service
Simon & Schuster
200 Old Tappan Rd.
Old Tappan, NJ 07675
800/624-0023
$169
One megabyte required.

Graphics Handler

From p. 16

functions in your input-event loop lets you properly handle special GadTools operations. These functions also return useful information in the Code field of the IntuiMessage. For example, the actual position of a scroller or slider is provided in this field, as well as the color selected from a Palette gadget.

When you finish your gadget list, you can dispose of the entire affair with a single call to FreeGList(), passing the original gadget list pointer provided to CreateContext().

GadTools are not only simple to create and use, they are also easy to enhance and modify. Because of the amount of information on GadTools gadgets alone, I did not go into GadTool's new menu facility (which greatly simplifies menu creation, see David Joiner's article, "Menus for a New Generation," p. 4, April/May '91) and some of the other features and factors of using GadTools. If you wish to dig deeper into the mechanics of GadTools, I suggest you study your 2.0 GadTools header files.

It is important to keep in mind that GadTools is not compatible with OS 1.3, so take care when using this powerful library. Ideally, you should provide a backup arrangement using standard gadgets. If you cannot do that, request 2.0 explicitly, and then exit gracefully if it is not available. If you intend to sell your software commercially or distribute it widely in the Amiga market, you would do well to consider 1.3 compatibility a must—at least for now. ■

Paul Miller has been a developer since 1985. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or via Internet (pmiller @vttcf.cc.vt.edu).*

Step Up To The Podium!

Admit it. You're an expert. You know how **it** works better than (almost) anyone. When you write code, you play by all the rules, yet your programs consistently out perform even those of the most wild-eyed ROM-jumping hacker. It's been obvious to you for some time that you should sit down at the keyboard, fire up your trusty text editor, and write an article explaining exactly how and why **it** should be done **your** way.

If the above description seems to fit you to a T, perhaps we should be talking. *The AmigaWorld Tech Journal* is looking for technical writers who have expertise in one or more areas and have a burning desire to share that information with the Amiga technical community. We need experts in all aspects of programming the Amiga, from operating systems to graphics to the Exec. You can write in any language you like—C, Assembly, Modula II, or BASIC. Best of all, you can include as much source code as you need, because all source and executable is supplied to the reader on disk. We also need hardware maestro's who can explain—in thorough detail—the inner workings of such complex components as the Amiga's chip set, expansion bus, and video slot. Don't forget algorithms either, we'll help you pass on your theories and discoveries.

The AmigaWorld Tech Journal offers you an unparalled opportunity to reach the Amiga's technical community with your ideas and code and to be *paid* for your efforts as well. So, whatever your "**it**" is that you want to write about, *The Tech Journal* is the place to publish it.

We encourage the curious to write for a complete set of authors guidelines and welcome the eager to send hardcopies of their completed articles or one-page proposals outlining the article and any accompanying code. Contact us at:

The AmigaWorld Tech Journal
80 Elm St.
Peterborough, NH 03458
603/924-0100, ext. 118

From p. 30

owchk() (in math.h), which creates a ray extending from the intersection point to the light source. It then checks to see if any polygons intersect the ray. (See Figure 12.)

If it finds an interesection, the function returns true indicating that the light is blocked and the point is in shadow. If the point is in shadow, the color is set to a very dark brown, otherwise it is set to a much lighter shade.

If a ground hit is not detected but polygon hits were, we call the shadepoint() function (in math.h) to shade the intersection point. Shadepoint() uses the vector dot-product operation to calculate the cosine of the angle between the incoming light ray and the intersected polygon's surface normal. The size of the angle determines the brightness of the surface at that point. In addition, the function checks if the point is in shadow. If so, the intensity of the surface point is reduced to account for the lack of light.

If we do not find a ground or polygon hit, the sky is visible for the pixel. In this case, tracer calls shadesky() (in math.h) to calculate a sky color. For effect, this routine shades the sky as though it has a different color at the zenith than at the horizon. Again using the dot product, the function calculates the cosine of the angle between the ray and the ground's normal. The greater the angle, the more the horizon color is used, and the smaller the angle, the more the zenith color is used. A simple linear interpolation mixes the two colors.

After the color for the current pixel is calculated, the program calls storeRGB() to store the color's RGB values in the global RGB buffers. To finish the tracing of the pixel, its color on the screen is turned to gray. The loop then continues to process the remaining pixels in the same manner.

As you can see, we have to deal with pixels only on an individual basis. This means that the entire ray-tracing process breaks down to the question, "What does this pixel's ray see?" It is the sophistication with which you answer this question that determines the quality of the resulting image.

When all the pixels have been traced and the tracing is complete, the program calls the last function, writeRGB() (located in write.c), and passes it the object's file name in the same manner as for the loadobject() function. WriteRGB() writes each RGB buffer as a separate file. Each file is given the same name as the object, with a .red, .grn, or .blu suffix appended to it. These RGB files contain the final image and can be viewed using View (in the Wagner drawer) or The Art Department (TAD) from ASDG. In TAD, separate red, green, and blue RGB files are referred to as the Sculpt format.

LOOKING GOOD

Although our trip was a simple one, you can make ray tracing more sophisticated in many ways. The best example is the shading function. A more powerful shading function would take into account variable surface attributes and textures. This, by itself, would create much more realistic images. In addition, you could add reflection and transparency/refraction to increase the quality even more. With a little ambition, you can turn out beautiful ray-traced images with your own program. ■

Brian Wagner, one of the founding members of Cryogenic Software, is the developer of 3-D Professional. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or via PeopleLink (CRYO) or CompuServe (72137,573).*

LETTERS

Flames, suggestions, and cheers from readers.

FILL 'EM UP

I liked your first issue, but I have a few suggestions for improving the disk.

1. Do not have icons that open to show an empty window.

2. Put a ReadMe file in each directory.

3. Include a contents file (a la Fred Fish) with page references to make locating the associated article easier.

These minor changes would make things much better, especially if the disk gets separated from the magazine.

Urban Ludwig
Laurel, Maryland

As you can see from this issue's disk (and the June/July issue's), we have made some changes. Drawers in the Articles directory are now labeled with the author's name and the page number of the related article for quicker reference between the text and code. Every drawer also has a ReadMe file (with an icon), listing the files the directory contains and other pertinent notes. The new ReadMe file in the root directory holds late-breaking additions and corrections that didn't make it into print. Finally, we moved Lharc into the main directory for easier access. If you can think of any other improvements, let us know.

SHARED STRIPS

In "Menus for a New Generation" (p. 4, April/May '91), David Joiner implies that having windows share menu strips is a quick way to get a guru. While this is generally true, there are special cases where it is false, such as when the two windows are also sharing ports. While making this work right is tricky, it is a worthwhile thing to know how to do.

Mike Meyer
Palo Alto, California

You *can* have two windows sharing the same menu strip, at least under 1.3 and below. I am providing two short programs as documentation for my assertion. (See the Letters drawer on the accompanying disk.) The first, MenuShare, opens three windows on a custom screen and attaches the same menu strip to each. The second, MenuList, displays all screens, windows, and attached menu strips known to Intuition. The address of the MenuStrip structure is displayed in hexadecimal format. I suggest you open a couple of disks and other windows on the Workbench screen. You will find that Intuition itself uses the same MenuStrip for each of its windows!

Fred Scheifele
Hamilton, New Jersey

I stand corrected. Menus can be shared if all the windows are managed by the same task and have the same UserPort.

David Joiner

BETTER DOCS, PLEASE

The lack of proper system documentation has been a great contributer to the failure of the Amiga to catch on. The Amiga is a difficult machine on which to program—not because of the language of choice (C), but because system documentation is inadequate. With an operating system that is ROM-based, changes in the OS invalidate much of the existing documentation, and new documentation lags too far behind the OS releases.

Too much of the "how-to" documentation for the Amiga consists of poorly translated European texts, often containing little content. Even the "real reference materials" fail to present all necessary information (such as library references, precedents of function calls, addressing modes, and so on), or present the information in an unorganized, hard-to-follow fashion. I know of many potential developers who simply gave up trying to write software for this system.

For the machine to succeed, more users are needed. To get more users, more quality software is needed. To get more quality software, more developers are needed. To get more developers, we must make it worth their time and effort to pursue an idea and turn it into an application.

Thomas R. Clark
Union City, California

LIBRARY LAMENTS

I hoped the article "Shared Libraries for the Lazy" (p. 30, April/May '91) would help me with my shared library project. Several times I tried unsuccessfully to build a shared library include file with Manx-style pragma statements as shown in the comment section of the example program Simple.c. If I modified the .fd file to show which register to use I always got an error message and no pragma statements. Only if the function definition had no arguments did LibTool write a pragma statment.

Alfred Steele
San Antonio, Texas

The proper procedure for using LibTool with register args and Manx 5.0 is as follows: First, alter the .fd file to include the desired register placement. For Simple.c, for example, the functions would look like:

```
Add2Numbers(num1,num2)(d0,d1)
Sub2Numbers(num1,num2)(d0,d1)
Mult2Numbers(num1,num2)(d0,d1)
```

Invoke LibTool as noted in the starting comment section of Simple.c. Remember to include the pragmas in the header file for both the library and the application, otherwise, the compiler won't know where you want the args placed.

Jim Fiore

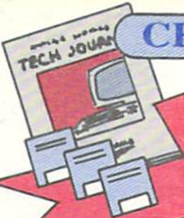
WHAT'S ON YOUR MIND?

We're ready to listen to your suggestions for the magazine or Amiga developers, complaints, or cries for help with technical issues. Just write to Letters to the Editor, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or speak your mind in* The AmigaWorld Tech Journal *conference (AW.TechJournal) on BIX. Letters and messages may be edited for space and clarity.* ■

CHARTER SUBSCRIPTION OFFER

SUBSCRIBE & SAVE

AVAILABLE ONLY FROM AMIGA WORLD



THE AMIGAWORLD TECH JOURNAL

☑ **YES!** enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of $59.95. That's a savings of $35.75 off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Canada and Mexico $74.95. Foreign Surface $84.95, Foreign Airmail $99.95. Payment required in U.S. funds drawn on U.S. bank. Available March 15.

Name ____________________

Address ____________________

City ____________ State ______ Zip ______

TS41I

☐ Check/Money order enclosed

Charge my: ☐ Mastercard ☐ AMEX ☐ Visa ☐ Discover

Account# ____________ Exp. ______

Signature ____________________

Or call 800-343-0728 ☎ 603-924-0100 for immediate service.

NEW! 6 BI-MONTHLY ISSUES PLUS 6 INVALUABLE DISKS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT #73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

The AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

The AmigaWorld

TECH JOURNAL

CHARTER OFFER!

✓ **YES!** Enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of $59.95. **That's a savings of $35.75** off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Name ______________________________

Address ______________________________

City ____________ State ______ Zip ______

TL411

☐ Check /Money order enclosed

Charge my: ☐ MasterCard ☐ Visa ☐ Amex ☐ Discover

Account# ______________________ Exp ______

Signature ______________________________

Canada and Mexico $74.95. Foreign Surface $84.95. Foreign Airmail $99.95. Payment required in U.S. Funds drawn on U.S. Bank. Available March 15.

Or call 800-343-0728 ☎ 603-924-0100 for immediate service.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

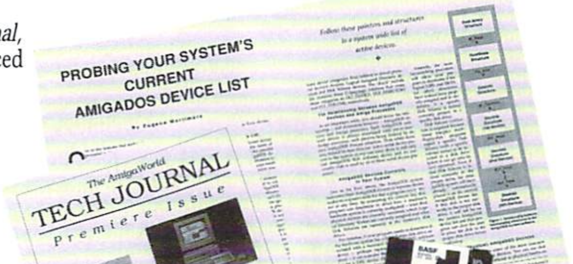
CHARTER OFFER

A source of technical information for the serious Amiga professional.

Introducing *The AmigaWorld Tech Journal*, the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bimonthly issue is packed with fresh, authoritative strategies and advice to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for good Public Domain programming tools on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!



Every issue includes a valuable companion disk!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include *some* technical information, but none devote *every single page* to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.

And only *The AmigaWorld Tech Journal* boasts of a technical advisory board comprised of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

Plus! Each issue comes with a valuable companion disk, including executable code, source code and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

The AmigaWorld
TECH JOURNAL

In every issue of *The AmigaWorld Tech Journal*, you'll find . . .

- Practical hardware and software reviews, including detailed comparisons, benchmark results and specs
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks
- Programming utilities from PD disks, bulletin board systems and networks
- Wise buys in new products—from language system upgrades to accelerator boards to editing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks. All at special Charter savings. Call 1-800-343-0728 or complete and return the savings form below—today!

To order, use this handy savings form.

Charter Savings Form

[✓] *Yes!* Enter my one-year (6 issues, *plus* 6 invaluable disks) Charter Subscription to *The AmigaWorld Tech Journal* for just $59.95. That's a special savings of $35.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal*, I'm entitled to receive a full refund—no questions asked!

Name ______________________

Address ______________________

City ____________ State ________ Zip ________

☐ Check or money order enclosed. ☐ Charge my:
☐ MasterCard ☐ Visa ☐ Discover ☐ American Express

Account No. ______________ Exp. Date ________

Signature ______________________

Satisfaction Guaranteed!
Or your money back!

Canada and Mexico, $74.95.
Foreign surface, $84.97.
Foreign airmail, $99.95.
Payment required in U.S. funds drawn on U.S. bank.

Complete and mail to:
The AmigaWorld Tech Journal
P.O. Box 802, 80 Elm Street
Peterborough, NH 03458

TJ891

For faster service, call toll-free 1-800-343-0728.

MEET THE PRESS

"The Toaster provides a wider range of functions than any combination of solutions found on (Macs or PCs) – and at a fraction of the cost. It could do for personal and corporate video-tape publishing what Gutenberg did for the printed word."
- Computer Currents

"Commodore's Amiga computer platform may have found the application it's been looking for with the Video Toaster."
- InfoWorld

"provides broadcast quality desktop video at a heretofore unheard-of price."
- Television Broadcast

"poised to join the camcorder and VCR as a home video appliance."
- Videography

"took the National Association of Broadcasters convention by storm"
- Electronic Media

"several networks plan to use the low cost system . . . Television will never be the same"
- TV Technology

"they could have doubled the price and still had to beat off potential customers."
- Broadcast Hardware

"One of the stars of NAB"
- Television Buyer

"NewTek was approached separately at Comdex by Apple and IBM, both offering NewTek whatever it cost to do a version of the Toaster for their respective systems."
- InfoWorld

"Both the talk and the toast of the video world, its impact will be roughly that of fire to food, the wheel to transportation, and the printing press to publishing."
- Christian Science Monitor

"it represents a significant new direction in home video's future . . . outperforms $100,000 digital effects units."
- Video Magazine

"the Video Toaster promises to toast dozens of high-end video gear makers into bankruptcy . . . it provides about $100,000 worth of power."
- John Dvorak, PC Magazine

"there may be lots of people who will buy the Amiga just so they can use the Toaster."
- AmigaWorld

"Bill Gates, the Microsoft Mogul, will be getting his very own Video Toaster."
- InfoWorld

"a significant price-performance breakthrough"
- Videography

"the first breakthrough product in the Amiga-video realm"
- AV Video

"This is going to sell a *lot* of Amigas . . . it will help drive the Amiga market as nothing else has."
- .info

"will forever change the way in which we communicate."
- Camcorder

"The Video Toaster will open up new markets for the Amiga"
- Computer Graphics World

"Something's burning, and it's not the Toast. It's the ears of network executives."
- Minneapolis Star Tribune

"Makes a personal computer act just like a Hollywood production studio"
- USA Today

"The equivalent of a $60,000 television studio"
- New York Times

"Coolest video/graphic product of the year"
- Byte

"a crusade to bring high-quality video functions to the mass market"
- AV Video

"The highlight of the recent Comdex trade show"
- John Dvorak in MacUser

"Ironically, our favorite hardware product of MacWorld Expo"
- MacWeek

"The Toaster creates desktop video the way Apple Computer and Aldus Corp. created desktop publishing"
- Los Angeles Times

"The hit of Comdex was neither PC nor Mac related. It was the Video Toaster"
- Washington Post

"as capable as gear normally costing $60,000"
- Business Week

"special effects magic"
- Success

"used by everybody from the guy who's entering America's Funniest Home Videos all the way up to MTV"
- Post

"The Video Toaster is the world's first desktop television studio."
- Computer & Business Technology

"the same functionality as professional video editing and switching equipment that costs from 10 to 50 times the price."
- PC Magazine

"made the biggest splash at this year's NAB . . . the product the Amiga has been waiting for."
- Videography

NEWTEK
INCORPORATED

1-800-843-8934