THE BANTAM AMIGA LIBRARY

AMIGA USER'S GUIDE TO GRAPHICS, SOUND AND TELECOMMUNICATIONS

DAVID MYERS

The Amiga User's Guide to Graphics, Sound and Telecommunications

To Dear Old Dad
—D.M.

For My Parents
—J.P.

The Amiga User's Guide to Graphics, Sound and Telecommunications

by David Myers
with programming by Joe Power

An International Publishing and Computer Services Book

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

The Amiga User's Guide to Graphics, Sound and Telecommunications
A Bantam Book / February 1987

Interior design by Mike Kelly, Slawson Communications, Inc., San Diego, CA.
Production by Slawson Communications, Inc., San Diego, CA.

Through the book, the trade names and trademarks of some companies and products have been used, and no such uses are intended to convey endorsement of or other affiliations with the book.

Every effort has been made to locate the copyright owners of material reproduced in this book. Omissions brought to our attention will be corrected in subsequent editions.

ISBN 0-533-34283-5

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

B 0 9 8 7 6 5 4 3 2

Contents

Part I

THE REVOLUTIONARY COMPUTER

CHAPTER 1

Introduction

The Commodore Amiga looks pretty much like any other home computer. It's a smallish box of white plastic, with a keyboard attached to the front, a series of plug slots at its back, topped by the ubiquitous monitor screen. No big deal. Right? Wrong. As soon as you see its color graphics, or hear its music on stereo speakers, or work with the multitasking features, or see how fast it does the jobs you want, you begin to realize that the Amiga is something different, a machine that is more than just another home computer.

The Amiga's capabilities are no accident. They are, instead, the result of deliberate designs that combine the best of existing technology together with innovative new concepts. Seeing where the Amiga sits in the historical progression of computer technology underscores, and helps explain, the whys and wherefores of its power and speed.

When electronic computers were first developed over forty years ago, they were monstrous machines consisting of thousands of vacuum tubes taking up whole floors of research laboratories and office buildings. The first programming of these behemoths involved rewiring individual circuits to make the proper connections for each program. Imagine having to rearrange a few thousand wires each time you want to run a new program.

Soon, however, a mathematician came up with the idea of installing permanent circuits that represented the basic elements of a programming language. Programming the machines then simply required turning on and off the appropriate circuits, and thus were born the concepts of operating systems and software that have evolved into today's programs.

This configuration of the computer made it vastly easier to program, but at the same time had an unexpected effect. Software began to develop quasi-independently of the hardware, leading computer designers to ponder which computing functions were best handled strictly as hardware and which were better served with software. The dilemma did not go away, even as computers became smaller and entered people's homes.

The first home computers were for hobbyists handy with a soldering iron. In other words, hardware was king. Producing software was a separate job undertaken once the hardware was in place. But as the hobby machines gave way to mass-market computers, the emphasis shifted as more functions became controlled by the growing number of software programs. In time, software usurped hardware as the more important part of the computer. Conventional wisdom at the time even suggested buying software first and then finding the computer that could run it.

For a while this focus on software worked fine, as the home computers did their jobs quickly, efficiently, and without much fuss. Then, slowly, an entirely predictable trend appeared. People, no longer intimidated by computers, saw the machines as simply better technology to get some work done and consequently wanted more speed, flexibility, and operating options from their computers. In response, software programs became longer, more complicated, and required more memory. And in response to those software characteristics, computers began to slow down. Instead of being a selling point, a computer's speed (or lack of it) became a sore point.

The stage was set for the Amiga. The Commodore Amiga swings the pendulum back to a balance of hardware and software that takes maximum advantage of both. It draws on the lessons learned with large computers — certain jobs are better done by hardware alone, while others can be accomplished best with software. By building a computer using the latest technology and designed with back-to-basics knowledge, Commodore has put development of home computers back on the main road. The Amiga's place in history is secure. And it may signal the start of future developments in computing.

THE FUTURE OF COMPUTING

Predicting the future of anything is always risky business, but trying to guess where an explosive technology like electronic computing will lead is like trying to hit a target blindfolded — while jogging. Nevertheless, certain trends in the development of computers seem to remain reasonably constant and can form the basis for seeing where computers will take us in the future. With a very deep breath, then, here are some predictions for computing's future.

Memory technology will continue to improve and get less expensive. RAM (random-access memory) technology is one of the brightest spots in the development of small computers. Originally expensive and hard to acquire, RAM chips are now so cheap and available that most computers come standard with at least 128K. Amiga comes with 256K. And the price per unit of memory will continue to decline as better production methods and new chip designs develop to meet the growing demand for more memory.

Of course, as the amounts of memory increase, so does the size of programs designed to take full advantage of the machine. Many business programs today require at least 320K (512K preferred). The tendency of software to push you to expand RAM to its limits will continue, but fortunately, the price of doing so won't be as expensive as before.

Mass storage — the disks and disk drives — will continue to improve, but, more importantly, a new breakthrough in technology will probably appear in the near future. The progression of mass-storage technology for small computers started with tape recorders, moved to 8-inch, then to 5¼-inch, and now stands at 3½-inch floppy disks. (Amiga's disks aren't really "floppies" because of their protective hard plastic casing. The actual disk inside the case, however, is the same material as the earlier floppies.) Disk size does not equate directly with storage capacity. Today, a single Amiga disk can hold 880K, which is as much as three times the amount of the 8-inch disks on earlier small computers.

Concurrent with the improvement in floppy disks, hard disks succeeded in capturing a large share of the market, to the point that now virtually every computer has an option for adding a hard disk. Hard disks typically have storage capacities in the megabyte (million byte) range. Like the floppies, hard disk size has decreased while storage capabilities have increased. That trend will continue.

The new mass-storage technology soon to appear for small computers is compact disks, or CDs. CDs, now used as video disks and audio compact disks, are derived from technology developed in the sixties. The disks look like shiny silver phonograph records that have a multihued sheen. Composed of aluminum coated with a plastic layer, CDs store information as a series of ultratiny holes, or pits, created by a laser beam. The beam is so precise that it separates the holes by only a few wavelengths of the laser light. Compared to recording on a floppy disk, which is limited by the width of the magnetic recording head, the density of data on a CD is orders of magnitude greater.

Reading the data on a CD is done by another laser beam which focuses on the plastic layer. The beam reflects back to a photosensor and is translated into the type of data stored on the disk. For video disks the data is the images of a movie; for audio disks the data is sound far surpassing the quality of any tape recorder or conventional record. For computers the data could be video, audio, text, or programs.

The promise of CDs is storage in the billions of bytes per disk (compared to millions of bytes on hard disks and "only" hundreds of thousands of bytes on floppies). One billion bytes is equivalent to about 450,000 pages of text. Other benefits of CDs are their virtual indestructibility and, once they become a commercial success, their low cost per byte of storage.

On the other hand, the main problem with some CDs is that they can be written on once, and only once, because the pits in the plastic layer cannot be erased. This, of course, is the complete opposite of the method you use today with floppies. On current floppy disks you can create a file, and later change, overwrite, or completely erase it. With the CDs a file remains there forever. For this reason, CDs for mass storage are also called "write-once" disks. (The disks themselves are called "WORM" for "write once read many.")

The limitation of "write-once" is not as bad as it sounds. Because CDs have the capability for so much storage, you simply "throw away" the space on the disk used up by old files and "keep" the space that holds the files you want to save. It sounds inefficient but in practice will be faster and more cost effective than the conventional methods of storing and editing files. An added benefit: thrown-away files are still available on the disk, so you can't inadvertently erase a file.

As it turns out, this write-once feature is one reason why CDs are now being touted more for ROM (read-only memory) than for mass storage. A single CD-ROM disk could hold all your programs, files, games, and other data. The computer could read, but not alter, the data on the disk, but you would still need some "writing" system, such as a standard disk drive for entering data.

Another reason for the focus on ROM applications is the prices of the two separate laser systems to read and write data. The laser mechanism for writing data on a CD differs from the one to read it. CDs used for mass storage require both a read laser and a write laser which substantially increases the cost of a CD storage system. Having CDs only for ROM means that individual computer owners would need only the read laser. (By the way, although not yet available to the general public, CDs for both reading and writing are in existence.)

Like the laser for CDs, other technologies will find applications in computers. Video cameras, for instance, will be able to feed images directly into memory where they can be digitized and displayed on the screen. You will be able to then edit the image by changing its colors or adding text to it. An innovative application of this feature currently in use on large computers takes in images of old silent movies and transforms them to color. If you've seen Fred Astaire and Ginger Rogers dancing in full color, you've seen the results of this remarkable technique. You'll be able to do similar things on your Amiga in the future.

Audio technology advances for small computers will not progress as fast as video. Synthesizers can now be connected to computers through MIDI (Musical Instrument Digital Interface), so you can play, record, edit, and control output with the computer's circuitry. Although the interest in MIDI is high, it isn't as pressing as is the demand for video technology. Audio output — the actual speakers — will probably stay with stereo for the near future. Quadriphonic sound was introduced a number of years ago and flopped. Apparently, not enough ears are discriminating enough to warrant the extra expense of better sound output.

Telecommunications will facilitate more networks and computer-to-computer interconnections. Networks aren't new. Neither are computer-to-computer connections. What the future does promise is more opportunities for networking, better modems, and communications software that is more flexible and easier to use.

Large networks, electronic bulletin boards, and electronic mail have not lived up to their initial billings, at least not for many owners of small computers. After an introductory rush of enthusiasm, the large national networks are growing at slower rates than expected and some are actually shrinking. No one knows, for certain, why this is the case, but one suspected culprit is the individuality of small computer owners. They seem to want more specialized information than the large networks are willing to provide. Supporting this claim is the growth of small, specialized, regional networks at the same time that the large ones are contracting. The trend to the small network connecting people with similar interests will continue.

Connecting different types of computers has always been a telecommunications bugaboo. To be sure, you could send files from a Commodore to an Apple, but only in ASCII format; all other formatting features were "disregarded." Technology, particularly software derived from large computer networks, is being developed that will make the differences between communicating computers transparent. As the software improves, you will have more flexibility in connecting with other types of computers, and the software itself will be easier to use.

Modems too will be easier to use. In the future the modems will communicate with each other automatically, set the necessary parameters for establishing the communications, and take care of all the technical aspects. Connecting with a network or other computer will then be no more difficult than making a telephone call.

Voice communication will also be added to the repertoire of the modems. In other words, you'll be able to send data to someone else and talk at the same time. This capability now takes two separate telephone lines or a single, expensive, high bandwidth line. In the future this capability will be part of a home or business standard communications system.

Computers will be integrated with other home electronic systems. With the exception of TV and stereo, home electronics is just in its infancy. Separate pieces of equipment have been developed independently of one another, leading to the current mix of machines in many homes. Consider, for example, that the data on a VCR tape is stored physically exactly as it would be on a computer disk. Only the different storage formats and the timing sequences used by each machine prevent a VCR tape from being linked directly to the computer as its storage medium. Similarly, a TV screen is essentially the same technology as a computer monitor, except that the number of scan lines on the TV is usually less than on the monitor, and the audio circuits are in the computer not the monitor. The electron gun that creates screen images, and much of its controlling circuitry, is duplicated in the two pieces of equipment.

The piece-by-piece approach to home electronics is similar to the days of component stereo systems when, to get the best possible system, you bought a separate tuner, tape recorder, turntable, speakers, and so on, wired all the pieces together, crossed your fingers, and turned on the power. But as designs improved, the quality of lower-cost, less-troublesome single system stereos began to rival that of the component systems. Nowadays most stereos are sold as complete systems in a single unit.

In the future, electronic components will be linked together to create a single "home electronics station." In it will be your Amiga, telephone, TV, stereo, VCR, satellite receiver module, laser disk reader, and jacks for adding the next electronic piece of equipment to be accepted in the marketplace. A single screen will be capable of simultaneously showing images from multiple sources, such as your TV and computer. A single storage source will save your VCR recordings and your computer data on the same disk, and the telephone will be able to send and receive data and pass it on to the other systems. Imagine receiving a film for your VCR through your telephone instead of going to the nearest video store . . . for a fee, of course, from the business sending the movie to you over the telephone lines.

The quality of computer graphics will continue to improve. Graphics from microcomputers have traditionally been noticeably more angular and chunky than most TV-quality images. The difference between them is due to the fundamental difference between TV signals and computer processing. The TV signal contains all the data an electron gun needs to create an image on each sweep of the screen. A computer, on the other hand, has to process data before it can control the gun, meaning that the processing has to be fast enough to get data in and out of the beam for each sweep.

Until the Amiga, inexpensive microcomputers simply could not process data for high-quality graphics fast enough and still keep up with the beam. Now that the Amiga has broken that "quality image" barrier, graphics from microcomputers will begin to rival those from large computers. At the same time, the technology for graphics, especially new TV screens such as high-definition TV, will be designed to take advantage of the new era of graphics.

Computer prices will stay about the same, but what you get for the price will increase. The computer business, like virtually any other retail business, has "pricing points" for its products. A pricing point is the amount people will pay for a certain type of purchase. Here, roughly, are the pricing points for the microcomputer market and how they are advertised to us consumers:

Under $100 — a game machine, but not suitable for any "serious" computing.

$100 to $500 — lots of games, some home computing, but not for business.

$500 to $1,000 — a good home computer, some work for small businesses, but not yet a business machine.

$1,000 to $1,500 — this is where most computers live, good for small business, homes, games, and even a few larger businesses.

$1,500 to $3,000 — for power users and "serious" computing.

$3,000 and up — the big-business market (surprisingly many of the machines in this price range are not much different from lower-priced machines, but the perception is that the higher price equates to more power).

Although these points fluctuate a bit as the market expands and contracts, they remain remarkably stable over the long run. What does change is what you get for the money. For instance, in 1975 the first real home computer (named Altair 8800 and made in New Mexico, not the Apple as many believe) cost about $500 in kit form, needed a separate monitor and floppy disk drives, cables, and assorted other equipment, which when all was said and done drove the price to about $1,500. What you got for that amount was about 16K of RAM, a single-sided 8-inch disk drive, disks capable of holding only about 160K, a black-and-white monitor, a small keyboard, and no software. Compare what you can get with an Amiga for that price.

The pricing points will remain reasonably constant in the future, but you'll get a lot more for your money. In other words, computer manufacturers will build more capabilities into a machine and charge the same price as its predecessors, instead of building a lesser machine and charging less.

Business users will continue to value reliability and consistency over technological innovation. Building a better mousetrap is no longer a surefire guarantee of success in the microcomputer market, especially if the computer is aimed at businesspeople. In fact, the opposite seems to be the case more often than not. The business segment of the market wants to be sure that an investment in a computer will remain a sound expense for a long time, and that the company selling the computer will be around in the future. The IBM PC is the classic case in point.

The IBM PC is not a particularly fast, powerful, or technically advanced machine. Its cost-to-performance figures are stunningly mediocre. Yet, even though it entered the market well after other manufacturers, the PC is without a doubt the best seller to businesspeople. The image of IBM, its stolidness and consistency, was, and still is, more important to businesses than technological marvels. For this reason, almost all microcomputers today have the ability to "look like" IBM PCs, despite having to slow down or degrade their other features in order to do so.

The trend of businesses to stay with IBM or other "established" companies will continue. In most cases this trend tends to stifle technological innovations and research. However, such is not the case with the Amiga. It fits all the necessary business categories: the Amiga comes from a respected, reliable company; it is technologically advanced; and it can emulate an IBM PC.

Gimmicks designed only to build sales will appear and disappear with regularity. Do you remember the Hewlett-Packard "touchscreen"? Instead of typing keys to select a command, you touched a portion of the screen with your finger. Small invisible beams around the edges of the screen determined the position of your fingertip and selected the command under it. A nice concept, but totally impractical. The beams were none too precise, so you were never completely sure that the command selected was the one you wanted. And the screen kept getting smudged.

Similar gimmicks are all too common for microcomputers attempting to distinguish themselves in the crowded marketplace. Although appearing less frequently than before (simply because there are fewer companies making computers these days), new gimmicks will nonetheless show their faces briefly and then disappear, much to the chagrin of the company introducing them. To people watching the microcomputer market, however, gimmicks are amusing testaments that some of the exuberance of the industry survives.

WHAT MAKES AMIGA UNIQUE

Amiga's uniqueness comes from two sources: its design and its mix of features. The Amiga's design includes three new custom chips that control graphics, sound, and animation. The chips also serve double duty by controlling I/O (input/output), direct memory access, the disks, and interrupts of the main CPU (central processing unit) chip. Relieving the CPU of many routine "bookkeeping" tasks is one reason the Amiga is so fast; it is also the key to the Amiga's multitasking capabilities.

The mix of features is unique because, for the first time, many of the most sought after microcomputer features are integrated into one machine. The philosophy guiding that integration is, "The best technology, with the highest quality, for the most reasonable price," or more succinctly, "The most bang for the buck."

The Amiga's features are what most people now want in a microcomputer: high-resolution graphics, animation, high-quality stereo sound, and an easy way to select options by using symbols (icons) on the screen. Although each of these features may be found on other computers, having all of them on one home computer is what makes the Amiga unique.

Here's an overview of the Amiga's main features:

The Amiga is the only microcomputer available today that comes standard with multitasking. Multitasking is the ability of the computer to run more than one program at a time — a feature traditionally found only on specialty microcomputers or on mini- or mainframe computers.

A series of "designed-in" features make the Amiga's graphics the fastest and sharpest available in today's microcomputers. The graphics features include up to 4,096 colors, 5 separate planes for creating colors, and 5 different screen resolutions. The screen has display options, including stacking 2 images on top of one another (called the dual playfield).

Animation subroutines are part of the ROM. Rather than drawing each individual picture as manual animation requires, programmers can animate a scene quickly by using the subroutines. The animation is supplemented by movable images known as sprites. The sprites are part of the hardware design. Other movable images are in the animation software subroutines.

The Amiga has "four-voice" stereo sound that can be independent of the main microprocessor. This means that you can create sound without interfering with other jobs (such as running animation sequences).

Another sound feature is synthesized speech. You can program the Amiga to speak in a variety of pitches, accents, and even (with a little effort) in foreign languages.

The Amiga has two user interfaces: Intuition and the Command Line Interpreter (CLI). Intuition is similar to the Apple Macintosh's method of selecting files and programs. You use icons on the screen to start an operation such as "Load a file." CLI is similar to the way IBM's DOS (disk operating system) works. Typed commands, such as DIR, instruct the computer to perform a function. Either of the options runs the Amiga.

The Amiga has two direct connections for expanding the system. You can add more memory or a series of other pieces of peripheral equipment. The machine's design has the capability for a maximum of 8.5 megabytes of memory.

CHAPTER

2

A Guided Tour

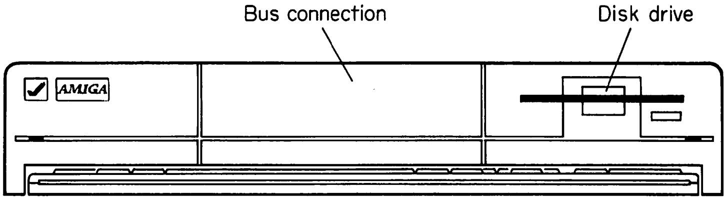
The Amiga is an advanced microcomputer, both inside and out. Chances are you'll never need to see the inside because the numerous plug slots and connectors on the outside provide complete access to the Amiga's interior. Nevertheless, knowing about the computer's layout and design can help you decide about adding peripherals and other equipment in the future.

THE PHYSICAL LAYOUT

The Amiga's facing panel contains two features (Figure 2.1): the internal disk drive and the bus connection for the addition of 256K of RAM memory.

Removing the middle of the panel reveals the memory expansion bus. A 256K memory expansion module plugs directly onto the bus and is then re-covered by the panel. Although this module is a standard add-on for the Amiga, it by no means exhausts the RAM expansion possibilities.

The panel on the right-hand side of the Amiga is where you plug in the mouse and other hand control devices (mostly for games) such as joysticks, light pen, paddles, and digitizer pad. You can attach two of these game devices simultaneously.



FRONT PANEL

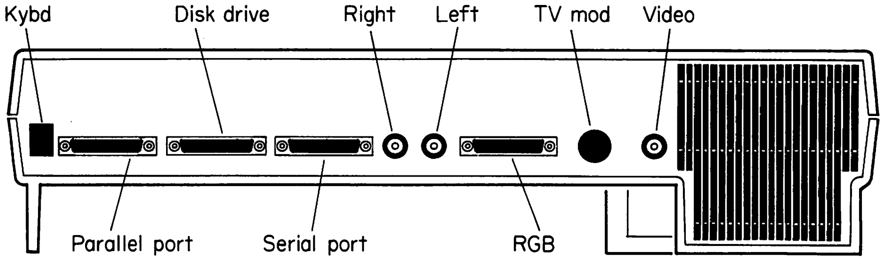
Figure 2-1

Further back on the right-hand side panel is the expansion bus for adding other peripherals. (More on this later.)

The back panel (Figure 2.2) contains nine ports for various peripherals. Your port choices are:

RGB monitor port — accepts both analog and digital RGB (red-green-blue) monitor. The Amiga passes three signals, one for each color, over the 23-pin connector cable, thus maintaining excellent signal separation. Digital TV also connects to this port.

TV port — sends signals to a TV set. Attach the TV set to this port uśing a standard RF (radio frequency) modulator cable. The Amiga's audio also passes



BACK PANEL

Figure 2-2

over the cable, but not in stereo. Both Amiga stereo channels are mixed to produce monoaural sound.

Video port — connects an NTSC monitor. (NTSC stands for National Television Standards Committee, although after comparing various models you may think it stands for Never The Same Color.) Collectively these monitors are called composite, because all of the signal is passed on a single wire and then separated in the monitor. Some have audio, but most do not.

Audio ports (2) — these stereo output jacks attach either to the audio input jack on your monitor (if it has one) or to separate stereo speakers. The ports accept standard RCA jacks.

Serial port — connects some types of printers and most modems. A separate Y-connector lets you switch between them. Use the special Amiga cable for the serial port.

External drive port — for connecting additional disk drives to the Amiga. You can connect up to three additional drives in daisychain fashion, but if you do, the second and third drives will need their own power supply.

Parallel port — connects most printers to the Amiga. Use a DB-25 male connector on the cable.

Keyboard port — attaches the keyboard to the Amiga.

THE PERIPHERALS

Aside from the CPU and internal disk drive, everything else connected to the Amiga can be considered a peripheral. You have a wide latitude of choices.

Monitor

Because of the Amiga's superior graphics capabilities, a good monitor is necessary to get the full flavor of the images. An RGB monitor is by far the best for the Amiga. Its colors are sharp and clear, and its images distinct. An RGB background has a deep black background which, compared to the gray background of TV screens, makes the colors that much richer. Essentially a scaled-down studio-quality monitor, the Amiga RGB monitor also has a speaker for audio output.

Obviously, this type of monitor produces the best images for business graphics and games. However, it also can clearly display 80 characters across the screen for use in word processing and other business tasks.

A monitor is also necessary to see the Amiga's full range of 4,096 colors and 2 high-resolution modes. With a TV set, you're limited to the 2 lower resolutions of the Amiga.

If you decide to shop around for an RGB monitor, be sure to specify that you want an ANALOG RGB monitor. The IBM PC and PC compatibles use a DIGITAL RGB monitor for graphics. This is not compatible with the Amiga and it only displays 16 colors. Recently, monitors like the NEC Multisync have appeared. These are capabale of handling both analog and digital RGB.

Similar to the RGB monitors are composite monitors. They can display in full color or monochrome (one color); the Amiga only works with the color versions. Depending on the monitor's quality, it can be either like a TV set or a "low-end" RGB monitor. Usually you can display the high-resolution modes and 80 readable characters for word-processing applications.

Your third choice for video is a TV set, but it is disappointing in light of the Amiga's capabilities. Not only are you limited to the lower-resolution modes, but the maximum number of readable characters on the screen is only 60 per line.

Another drawback of TV video has to do with the way a TV set shows pictures on the screen. TV images are not as intense as those on monitors because the phosphor backing on the TV tube has the tendency to hold an image as an afterglow. The more intense the image, the longer the afterglow. For normal TV viewing, an afterglow is irritating because it makes scenes linger on the screen and blurs the image, but for viewing characters on the screen, an intense image improves readability. Thus, TV images make letters and numbers inherently less readable than those produced by monitors with intense images.

A final TV disadvantage, although minor, is that you're limited to "only" 3,616 of the full 4,096 available colors.

Disk Drives

Amiga internal and external disk drives designed are for 3½-inch floppy disks enclosed in a hard plastic sleeve. The drives read double-sided disks, each of which holds up to 880K (about 580 pages of text). Each external drive has a connector for an additional drive; you can connect up to a total of three external drives.

You can also connect a 5¼-inch drive which uses standard floppy disks in flexible sleeves. Typically you would attach a 5¼-inch drive to transfer files and use programs developed on another computer. Additional software is required to run the drive. Hard disks are also available for the Amiga.

Audio Speakers

As described more fully later, the Amiga produces four channels of sound. Two channels are assigned to each of the two output jacks on the back panel. Using a Y-connector, you can run all four channels through a single speaker, such as the one in the Amiga RGB monitor, but more likely you'll want to run them to individual stereo speakers.

Printers

A number of printers attach to your Amiga. The drivers for the following list of printers are all standard in the Amiga:

Alphacom Alphapro 101
Apple LaserWriter
Brother HR-15XL
Commodore MPS 1000
Diablo Advantage D25
Diablo C-150
Diablo 630
Epson JX-80
Epson FX series
Epson RX series

Hewlett-Packard LaserJet and LaserJet Plus
Okidata Okimate 20
Qume LetterPro 20

Other printers connected to the Amiga require their own individual drivers.

EXPANSION POSSIBILITIES: OPEN ARCHITECTURE

The Amiga was designed with expansion in mind, a philosophy that harkens back to the early days of microcomputers. The first microcomputers' insides were mostly empty space, consisting of open slots for additional printed circuit boards. A computer owner had direct access to the slots by simply removing the top of the computer. Moreover, the slots encouraged independent developers to design, produce, and sell boards that plugged into the slots, thus expanding the capabilities of the basic computer. In that way, computer owners could customize their machines to have the applications they deemed most important. It also meant that the computer manufacturers could keep down the price of the basic computer.

Typical add-on boards provided computers with additional memory, graphics capabilities, clocks, internal modems, special video circuitry, and a host of other features. Some of the earlier computers almost demanded an extra board for any useful business work. The early Apple computers, for instance, needed an extra board (called an 80-column card) to produce 80 columns of text on the screen. Today, many of the features relegated to the extra boards are now standard in microcomputers, which leaves room for adding on more advanced features.

Physically connecting the boards was not too difficult for someone handy with a screwdriver. Simply removing the computer's cover revealed the slots; the boards slid into them easily and connected with wires and cables to the rest of the computer. Occasionally, boards required a resetting of the computer's DIP (dual in-line package) switches, an irritating but relatively simple chore. The "electronic connections" — the signals and bit streams within the computers — were also usually designed for adding the extra boards. Developers could build the boards using standard conventions for the electronics. Similarly, the computer manufacturers made memory and register addresses, and other programming features, available and accessible to the developers.

This easy access to a computer, both physically and electronically, is known as *open architecture* and is a prime reason people, and especially businesses, have embraced microcomputers so readily. One key factor in evaluating which computer to buy, in fact, has traditionally been the number of slots available for expansion. However, for reasons known only to themselves, a number of computer manufacturers have abandoned this time-tested philosophy. The Apple Macintosh and Atari ST computers have closed architectures, with Apple going so far as to void a Mac's warranty if the computer cover is removed. And attempts to open up the Mac and ST after the fact have been stopgap at best. The Amiga's design, on the other hand, says expansion and open architecture at every turn.

The Amiga's memory has over 16 million addresses. About half of the memory — 8.5 million bytes — is allocated to RAM, and most of the rest is reserved for "future use," with specific addresses reserved solely for expansion hardware. Furthermore, the part of the memory reserved for the expansion hardware establishes a protocol for designers which, if followed, will make the Amiga automatically accept the hardware without the slightest modification. Even the DIP switches will not need resetting.

Adding boards has been made simpler on the Amiga. Instead of having to open up the computer and plug them in, additional boards (and other hardware) attach directly to the expansion bus on the right panel. The bus is a 60-pin connection that provides complete access to the inside of the machine, including the main processing chip and the three Amiga custom chips. Expansion boards or other hardware for the Amiga will simply slide onto this bus. Like adding more external disk drives in a daisy chain, you can keep adding as many hardware packages as possible, until the memory limits are reached.

Keeping the expansion boards physically outside but electronically inside solves two technical problems that plague many microcomputers. First is heat. Computers generate heat, and the more boards, the more heat. But heat is the kiss of death to sensitive electronic components. Microcomputers packed with expansion boards have a higher than normal breakdown rate simply because the cooling fan inside the computer cannot dissipate the heat fast enough. Putting the expansion boards in a separate box outside of the main computer keeps this problem from occurring.

The second technical problem solved by the external setup is failure of the power supply. Boards inside the computer draw their power from the same source as the computer itself. Computer manufacturers take this into account and design their power supplies to handle a reasonable demand for power by both the computer and expansion cards. But the manufacturers have no way of knowing what kind or how many power-hungry boards computer owners will add. The result is persistent failure of power supplies. The Amiga's expansion hardware will draw power on its own, relieving the internal power supply of excessive demands.

The numerous ports on the Amiga also indicate expansion. They connect to different types of peripherals you can add to your computer.

What kind of expansion possibilities? Of course, the Amiga accepts the usual add-ons of memory, modems, and math processors. But you'll also be able to add advanced microcomputer hardware not available for other machines. Some samples are:

- MIDI interface. MIDI, the Musical Instrument Digital Interface, is the means to attach a musical synthesizer to a computer. You can then program music on the computer and play it back through the synthesizer.
- VCR or video camera interface. Using a genlock card, you can pass signals from the VCR or video camera to the Amiga to produce digitized images on the screen. Once captured in the Amiga memory, the images can be part of other pictures that you create with graphics programs.
- CD (compact disks). The CDs can be additions to ROM or expansions of memory.

Expect other advanced add-ons in the future.

INSIDE THE AMIGA

Getting inside the Amiga is not too difficult. Remove the mounting screws from the top of the case and lift it off. Unscrew the metal shielding but do not try to lift it off yet. First, locate the four tabs around the bottom of the shield and gently bend them with a pair of needlenose pliers so they fit through the small slits on the motherboard. Now lift the shield up and off and place it aside.

The Amiga's central processing unit (CPU) chip is the Motorola 68000. It is classed as a 16/32 bit processor, meaning that it can process instructions 32 bits long, but passes data on the data bus in 16-bit segments. Perhaps the most innovative feature of the Amiga is the three custom chips that enhance the operations of the 68000. The three chips control animation, graphics, and sound, but they also handle other internal processing functions.

The *animation chip* acts like an assistant to the 68000. It contains a coprocessor, called the Copper, that controls the output of the other two custom chips relative to the screen position of the video beam. In particular, the Copper controls the output of graphics and sound, keeping the audio and video synchronized on the screen. In other computers this function is the responsibility of the main processing chip which slows down the computer's overall processing speed considerably. Having the Copper handle the work relieves the 68000 of the task, which translates into a faster-acting computer.

A separate section of the animation chip contains the Blitter, which controls line drawing, image movement on the screen, and coloring of bounded, or closed, areas. The term Blitter is derived from the jargon term "bit-mapped block transfer," which is another way of saying movement of screen images.

An additional responsibility of the animation chip is to control direct memory access (DMA). DMA is a process whereby different portions of the computer, including peripherals, can directly access memory without going through the 68000. This relieves the 68000 of many routine chores, like finding the address of a specific piece of data requested by the other portion of the computer. The Amiga has twenty-five DMA channels dedicated to features such as the video output, audio output, disk drive, and color control.

The custom *graphics chip* controls the screen display and keeps track of the relative position of the images on it. Output to the RGB, TV, and video ports comes directly from this chip.

The *audio chip* controls all four channels of sound as well as the disk controller and the interface for the serial port and the mouse/joystick port. In addition, the audio chip directs the interrupts to the 68000 from fifteen sources, including all the peripherals and the sound channels. In other words, a peripheral sending a signal to interrupt the 68000 sends the signal to the sound chip. The sound chip first determines the priority of the interrupt and then transmits the interrupt request to the 68000. After processing any interrupts of higher priority, the 68000 attends to the new request. The ability to handle interrupts in this fashion is a primary reason why the Amiga can do multitasking — the processing of more than one task at a time.

MORE ABOUT DIRECT MEMORY ACCESS

Much of the Amiga's speed is due to the design and use of the DMA channels relative to the computer's operating speed. The 68000 operates at a frequency of 7.15 MHz (megahertz, or millions of cycles per second), while the memory and internal bus operate at twice that speed, or 14.2 MHz. This means that roughly half of the cycles when the memory is available are not required by the 68000. In the Amiga, those "unused" cycles are allocated to DMA.

In its simplest form the DMA works like this: the 68000, Copper, and Blitter can access memory on all the even cycles of the operating frequency; the DMA requests get to access memory on the odd cycles. Each time the cycle is an even one, the 68000, Copper, or Blitter gets some data, or puts it on the appropriate internal bus, or runs an instruction. Most of the 68000's time alternates between running internal calculations and putting data on the bus. Thus, for instance (forgetting the Copper and Blitter for the moment), at cycle 0 the 68000 would put data on the bus, at cycle 2 it would perform a calculation, at cycle 4 it would put a result on the bus, and so forth.

During the odd-numbered cycles, DMA requests from various peripherals or other parts of the computer get the data *they* want from memory and put it onto the bus. The DMA processing bypasses the 68000 completely and handles tasks such as outputing sound to the audio channels, sending data to the disk drive, and moving a screen image to a new screen position.

This sharing of the system is the secret to the Amiga's speed. Because the memory speed is twice that of the 68000, and because the 68000 only accesses memory on every other cycle, the 68000 operates at its full speed even though other processes are occurring simultaneously. The DMA is essentially transparent to the 68000 and thus the computer is, for all intents and purposes, doing two (or more) things at once. It is why the Amiga can read a disk, play music, and display animation without visibly slowing down.

Of course, there are times when the DMA and 68000 may need to infringe on the other's cycles. This occurs, for example, when the video screen is packed with images or is using a large number of colors. The DMA channels "steal" cycles from the 68000 to process the video data.

The 68000 can also lose cycles to the Copper and Blitter as they draw images or keep track of animation on the screen. However, the Copper and Blitter are very efficient at doing their tasks. If the 68000 tried to draw each image itself (as is done in other computers), it would take longer even if the 68000 was not interrupted at all. Consequently, the priorities among the Copper, Blitter, and 68000 optimize the use of the computer and deliver the extremely fast Amiga performance.

When competition among the DMA channels, Copper, Blitter, and 68000 does slow down the performance of the Amiga, extra memory or judicious memory management can sometimes restore the better performance.

MEMORY MANAGEMENT

Memory management is a programming task to free as much internal memory as possible when it isn't needed. The more free memory available to a task, the faster the Amiga can process it. Three areas of memory can be managed by programming, although different programming languages and options employ different management methods. The three areas are: stack, heap, and BASIC's data segment.

The stack memory is for keeping track of local parameters and return addresses. For example, if a program has a subroutine call to another instruction, the memory address of that instruction is in the stack. The heap memory contains space for such things as static variables. For example, when defining the type of sound you want from one of Amiga's four audio outputs, the definition is stored in the heap. Each output description takes 1024 bytes of RAM in the heap. BASIC's data segment portion of memory contains the text of the program, numeric variables, strings, and room for data buffers.

In AmigaDOS, you can use the STACK command to set aside a certain size of the stack, in bytes, for use with DOS commands. The STATUS command tells you how much stack is in use.

One method of conserving space in BASIC is to keep the number of levels in your programs to a minimum. For example, "nested" subprograms each use stack space to keep track of their addresses, and the fewer the number of levels for the nested subprograms, the fewer addresses in the stack. Similarly, to conserve space in the BASIC data segment, link small programs in a series instead of nesting them in a single large program. Another data segment conservation method is to assign variables integer values instead of single- or double-precision numbers.

Conserving space in the heap is an easy memory management procedure. Simply release the buffers allocated to the specific commands using the heap. In BASIC the commands SOUND, WAVE, LIBRARY, WINDOW, and SCREEN all use the heap. To release a buffer for a WAVE command enter a WAVE 0 line at the end of the program. The other commands use similar procedures to release the buffers in the heap space.

The CLEAR statement in BASIC allocates a specific number of bytes to all three memory locations. To find out how much memory is currently allocated to the stack, heap, and data segment, use the FRE statement in BASIC.

Managing memory is not an absolute requirement on the Amiga. You can program and run routines without giving memory a second thought. Managing it just makes the Amiga more efficient and responsive to your instructions.

CHAPTER 3

Using the Amiga

So there it sits. You've bought the Amiga, hooked together all the peripherals, plugged it in, and are ready to go. Now what? The first step is to get familiar with the fundamentals of operating the Amiga. Then you can decide on the programs to do the things you want.

A QUICK WORD ABOUT PROGRAMMING

You don't have to program the Amiga yourself. Many commercial programs available for the Amiga create amazing graphics, produce wonderful sound, play new and entertaining games, and accomplish a complete range of business tasks. The choice is yours. On the other hand, programming the Amiga customizes it for the specific tasks you want it to do.

The rest of this chapter — and the rest of this book — is devoted to explaining how to use and program the Amiga yourself. To be sure, the book's programs are not as detailed as the commercial versions that caused professional programmers hours of sleepless nights. Nevertheless, they do show you some of the Amiga's capabilities, and at the same time open the door for embarking on more exotic work if you find it stimulating.

The programs in this book are in BASIC, specifically, Amiga BASIC. Amiga BASIC has commands and procedures designed to take advantage of the Amiga's unique graphics and sound capabilities, as well as its multitasking features. The process for programming in Amiga BASIC, however, is much the same as programming in BASIC on any other computer. Think of Amiga BASIC as an expanded "dialect" of BASIC.

Simply for efficiency's sake you should have at least a nodding acquaintance with BASIC before attempting to follow this book's programs, because explanations of each program assume you understand BASIC's concepts. For example, a description of a loop explains what the loop is for, but not what a programming loop is or does. If you aren't familiar with BASIC, a brief primer book or course would be helpful. The first few chapters of the Amiga BASIC manual also describe some of the necessary concepts.

Why BASIC and not the programming language C or the Amiga's machine language? Because BASIC (for all its faults) is still the most popular programming language for both amateur and neophyte programmers. You could, of course, program the Amiga in C or machine language. C is a powerful language, but difficult to learn and even more difficult to use proficiently. Machine language gets the most out of the Amiga in the most efficient way, but if you can program in machine language, you're probably developing commercial programs for the Amiga anyway. There is one other reason for Basic: It comes free with each machine. This means that everyone has a standard language to work with.

AMIGA INTUITION

The icons and visual system that you use to interact with Amiga's operating system is known as *Intuition*. Intuition controls the basic processes of reading disks, displaying characters on the screen, and the other necessary routine chores done by the computer. The Workbench disk and the Amiga "user interface" are also part of Intuition. A user interface contains the commands and images for interacting with the computer to make it start or stop a job.

Amiga's user interface is similar to the one pioneered by Apple's Lisa and Macintosh computers (which borrowed the basic concept from Xerox in the first place), but the Amiga user system extends Xerox's and Apple's versions to a new and unique interface that, as its name suggests, is intuitive to use. No commands to remember or typing required. Unlike other computers that put their operating systems in ROM, Intuition is stored on the Kickstart disk. This enables you to make changes to it or get later upgraded versions of Intuition in the future.

To start the Amiga, insert the Kickstart disk, let it load, and when an image of the Workbench disk appears, insert that disk. The screen then displays the Amiga user interface. Four basic items make up the interface: the mouse, icons, windows, and menus.

The Mouse

The mouse has two functions: it moves a pointer on the screen and it initiates commands to the computer. As you slide the mouse around, the pointer moves in a corresponding manner on the screen. Depending on the programs on the Amiga, the pointer can take on many shapes, such as an arrow, cross hairs, simple bar, or human hand. It can also be reprogrammed to adopt other descriptive shapes, such as a paintbrush for a graphics program.

Mouse or Keyboard?

Some people object to using a mouse. Whether it is because a keyboard is more familiar, or that the name conjures a furry creature (and who would want to hold that in a hand), the mouse is, for some, a real psychological dread. For them, Amiga has developed keyboard procedures that replicate mouse operations.

To position the pointer from the keyboard, press either the right or left Amiga key (the hollow or filled A) and an arrow key. The mouse moves in the direction of the arrow. To increase the speed of the pointer, simultaneously press the three keys: Amiga key-Shift-arrow.

To duplicate pressing right mouse button, simultaneously press Right Amiga-right Alt.

To duplicate pressing the left mouse button, simultaneously press Left Amiga-left Alt.

To select screen items using the keyboard, hold down the appropriate key combinations, such as Right Amiga-right Alt, and then press the arrow keys until the pointer is on the screen item you want to select. Release the keys to select the item.

Two buttons on top of the mouse *select* commands for the computer. Selecting a command tells Amiga to do the job that command controls. Putting the pointer on screen images selects commands represented by icons or by menus. The right button on the mouse selects menu choices, the left button selects icons and other functions. Pressing and releasing a button while the pointer is on an icon or menu is known as "clicking" on the item. The terms clicking and selecting are usually interchangeable.

Icons

Icons are small symbols on the screen that represent Amiga operations. There are four types of icons (Figure 3.1):

Disks — look like small Amiga disks. Selecting a disk icon displays other icon choices.

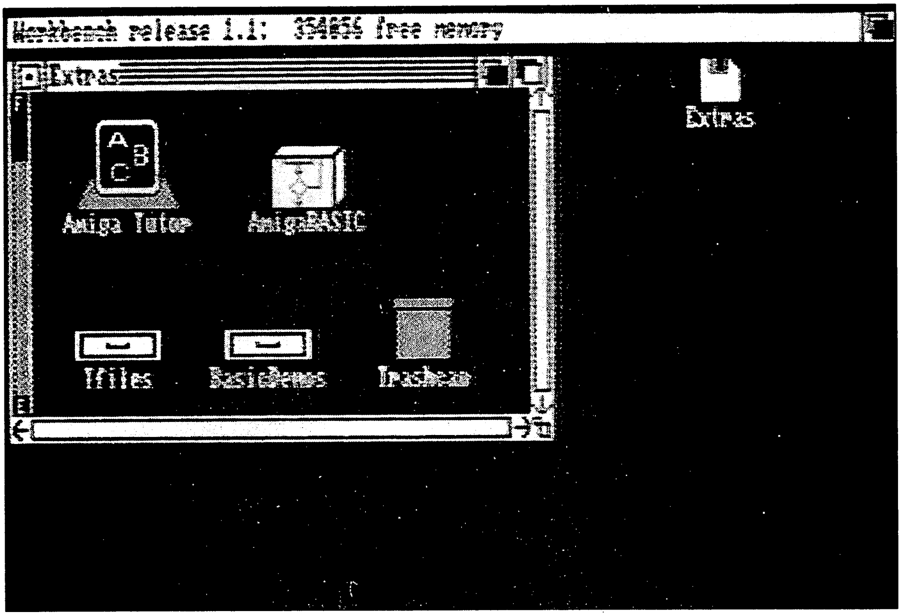


Figure 3-1

Tools — can be any shape. Tool icons start the programs represented by the icon. ("Tools" is another word for programs.)

Projects — can be any shape. Projects are files created by a program.

Drawers — look like small desk drawers. Drawers are directories where you store projects, tools, and other drawers. A trashcan symbol is a special drawer for deleting data from a drawer.

When you load the Workbench disk, the only icon on the screen is a disk in the upper right corner. Putting the mouse on the disk and clicking the left button twice selects the disk. This *opens* the Workbench disk icon (Figure 3.2) and displays a set of seven icons. You can then open any of these icons by putting the pointer on it and clicking the button twice again. When you open an icon, a *window* appears.

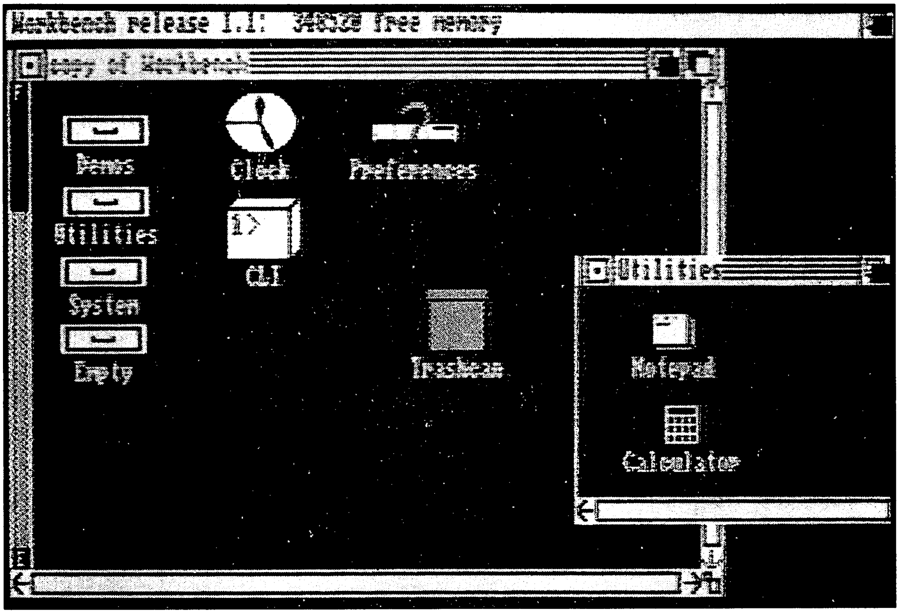


Figure 3-2

Windows

Intuition's windows are where you see information or images that an icon represents. For instance, selecting the icon labeled Preferences displays images for customizing the Amiga according to your own preferences. The images include selections for setting the time, date, screen colors, printer type attached to the machine, graphics, and so on.

Each Amiga window has features to control it. These features are known as gadgets, bars, requesters, and alerts.

Gadgets — symbols that control a specific window function. Select a gadget by clicking on it.

The close gadget closes the window. The front or back gadgets move the window in front of or behind other windows on the screen. The sizing gadget changes the size of the window. Changing a window's size uses a mouse operation known as *dragging*. Put the pointer on the sizing gadget, press the left button but do not release it. Still holding the button down, move the pointer to another position on the screen. Release the button. The window's size now extends to the new position of the pointer.

Other screen items respond to dragging. Icons can be dragged to other screen positions, and some graphics programs use dragging to draw lines. A window's bars also require dragging.

Bars — horizontal and vertical symbols for moving the window or data in it.

The drag bar located at the top of a window is for moving the entire window on the screen. Drag the drag bar around the screen and the window follows it. In this manner, you can position windows wherever you want them on the screen.

The two scroll bars move information within a window. The bars are on the left and bottom edges of a window. Drag the horizontal bar to scroll information horizontally in the window; drag the vertical bar to vertically scroll information. Click on the arrows at either end of the scroll bars to see a complete new window of information.

Requesters — boxes containing messages and choices asking for your response.

Requesters appear when a program wants you to do something or confirm some action. A typical requester is the message "Please Insert Disk X." For confirming an action, many of the requesters have small boxes labeled "OK" and "STOP." When the Amiga requires you to confirm a step in a program, a requester "pops up" on the screen. If you want the step to occur and the program to continue, click on the OK box; otherwise, click on the STOP box. The program will not proceed until you click on one of the two boxes. Requesters are also known as dialogue boxes.

Alerts — emergency messages indicating that you must attend to something before doing anything else with the Amiga. Alerts preempt all other Amiga programs and operations.

Due to Amiga's multitasking capability, the screen can show a large number of windows simultaneously. Each task or program must have its own window while it's running, but only one window at a time can be

Windows Are Virtual Terminals

When a standard computer — one that does not have multitasking — is sending data to a printer or some other terminal, the entire computer is devoted to that task. You get a coffee break until the printing or other work ends.

On the Amiga you can run many tasks at the same time. To accomplish this feat, the Amiga sets aside specific areas in memory for each task and automatically keeps track of which task corresponds to each memory area. Then, input to the memory area and output from it apply only to the specific task's window. In essence, the Amiga is treating each window as if it were a separate terminal, which, in computer jargon, makes each window a *virtual terminal*. In other words, the window isn't really a terminal but merely appears to be to the Amiga.

Appropriately enough, the term *virtual* is adapted from the study of mirrors. Mirror images, known as virtual images, are simply reflections in glass but "trick" your eyes into appearing as real objects. Likewise, the memory area "tricks" the Amiga into seeing it as a real and separate terminal. With this capability, the Amiga can run multiple tasks simultaneously, and each task thinks it has access to the entire computer. Perhaps a better explanation is: "It's all done with mirrors."

active. The active window is the only one that can receive inputs from the mouse or keyboard. Programs in the other windows cannot receive input, but they can still be processing data or performing other tasks.

To make a window active, simply click anywhere in it with the left button. The active window appears darker than the inactive windows. Closing a window with a close icon makes the last previously active window, the new active one.

Menus

Intuition menus appear as words (or titles) across the top of a window. Sometimes to keep a window uncluttered, menus are hidden; click the right mouse button to see a window's menus.

Each menu usually contains a list of choices or selections. To see those selections, put the pointer on the menu word or title and press the right mouse button. The menu's selections "pull down," or "unroll," on the screen. Drag the mouse to the selection you want and release the mouse button. This selects that menu choice and starts it. Sometimes a selection has its own choices (submenus) and you repeat the process selecting from them.

Menu selections control specific functions for its window. For example, if the window is a graphics program, one menu item may be Color. Selecting Color then displays a series of options for changing the color of an image drawn in that program. Similarly, for a sound program, a menu item may be Volume; selecting it lets you vary the volume of the sound produced by that program.

Each program and each window has its own menus. Selecting a menu item in one program's window will not affect the processes in other windows. In some programs, menu items can only be selected when certain conditions are present. For example, a program with a menu option for modifying an image on the screen may require that you save the image in a file first. The menu option for the modification will be displayed in light tones and will not respond when you select it. But as soon as the original image is safely stored in a file, the menu option appears normal and can be selected.

USING AMIGADOS

The AmigaDOS (disk operating system) Command Line Interface, known as CLI, is an alternative to Intuition. Whereas Intuition uses icons and images to select commands, CLI uses typed commands similar to IBM's operating systems known as MS-DOS or PC-DOS (see Figure 3-3).

CLI has forty-five commands for doing the routine tasks of disk copying, listing files, seeing the file directory on a disk, and so forth. Some of these commands are very similar to the ones used on the IBM PC. For example, the command DIR shows the directory of files on the disk in the current drive, and CD (for change directory) sets the current directory to a new one. These commands are the same in CLI as in MS-DOS.

In addition to the routine commands, CLI includes those that control some of the Amiga's unique capabilities. The CLI command SAY, for instance, makes the Amiga speech synthesizer speak to you; the AmigaDOS NEWCLI command starts a multitasking job.

CLI is stored on your Workbench disk. To use AmigaDOS, you must first leave Intuition and then start the Command Line Interpreter. CLI changes the screen from icon-oriented to command-oriented. You can then type in AmigaDOS commands. To start CLI, load the Workbench disk and select (double-click on) Preferences. The Preferences screen that appears has a box at its bottom left labeled CLI, followed by two choices: On and Off. Select On. If you want to use CLI all the time instead of Intuition, select the box at the lower right of the screen labeled Save. This saves your selection on the Workbench disk. Then, the next time you load Workbench, the CLI screen appears. If you don't want to save your On selection, select the Use box. Now select the System drawer from the Workbench and select the cube labeled CLI.

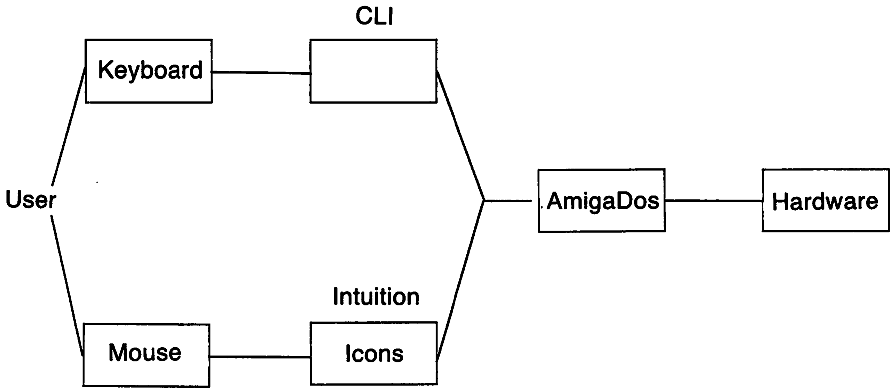


Figure 3-3

The window that opens has a 1› for a prompt. Although it appears similar to an A› or C› prompt on an IBM machine, the 1› tells you that this is the first CLI window you've opened. (The A› and C› prompts on an IBM refer to disk drives.) If you open a second CLI window, its prompt will be 2›. When the prompt appears, you're ready to control your Amiga with AmigaDOS.

One operating difference between AmigaDOS and IBM's DOS is that AmigaDOS is not stored in the memory; it remains on the disk until you type one (or more) of the commands. The Amiga finds the command, loads just that command into memory, and then runs processes you specified. This means more of the memory is available for processing; it also means that the Workbench disk must stay in the drive in order for the Amiga to access the commands.

Intuition or CLI

With its descriptive icons and mouse-activated selections, Intuition is easy to learn, easy to use and remember, and gets the job done quickly. The selections on Intuition, however, are limited to the ones you use in day-to-day computer operations. AmigaDOS takes more time to learn, you have to remember a set of commands and their variables, and the commands take longer to type than simply clicking on an icon. But AmigaDOS gives you more options and greater control over the operations of your computer.

Which to use? More often than not, you'll find that the particular job you're doing dictates which one you use. That is, starting a program or saving a file are fast, easy jobs with Intuition, while creating a multilevel set of directories is better suited to AmigaDOS. Take your pick.

The following table lists AmigaDOS's commands and provides you with a brief description of each. For complete instructions about AmigaDOS, see the manual entitled *AmigaDOS* available from Bantam Books.

Command	**Function**
Alink	Puts separate pieces of a program into one file that can then be run as one program
Assem	Assembles programming code for the 68000 chip
Assign	Assigns an Amiga device name (such as a disk drive) to a file directory
Break	Interrupts a program
CD	Changes the current directory or drive
Copy	Copies files from one disk or directory to another
Date	Displays or changes the setting for date and time
Delete	Erases files or directories
Dir	Lists files on a disk or in a directory
Diskcopy	Copies an entire disk to another disk
Download	Loads programs into the Amiga from other computers
Echo	Adds messages or commands to an existing AmigaDOS command
ED	Starts a full-screen editing program
Edit	Starts a line-by-line editing program
Endcli	Ends working with CLI
Execute	Runs a set of commands already programmed and saved in a file
Failat	Ends a program when certain conditions are met
Fault	Interprets error messages
Filenote	Adds a comment to a file
Format	Prepares a disk for working with AmigaDOS
If	Tests if a condition in a program is true or false

Command	Function
Info	Displays status of disk drives
Install	Puts programs on a disk so it will start automatically
Join	Merges up to fifteen files into one file
Lab	Adds a label in a program (used with the Skip command)
List	Displays names and status of files and directories on a disk
Makedir	Creates a new directory
Newcli	Starts a new CLI window
Prompt	Changes the prompt messages that appear on the screen
Protect	Assigns codes to a file that can keep it from being deleted, replaced, run, or read
Quit	Ends a program
Relabel	Changes the volume name of a disk
Rename	Changes the name of a file or directory
Run	Runs a program
Say	Translates typed text into synthetic speech
Search	Looks for a word or phrase in files and directories
Skip	Makes a program jump to a position in the program held by a label
Sort	Alphabetically sorts text in a file
Stack	Displays or allocates the amount of stack memory
Status	Displays status of all the tasks in progress
Type	Displays the contents of a file on the screen
Wait	Suspends a program for a period of time
Why	Explains why a command failed

USING AMIGA BASIC

Amiga BASIC is stored on your Amiga Extras disk. To start Amiga BASIC, insert the Extras disk into the drive and when the Workbench screen appears, select the Amiga BASIC icon. Unlike most versions of BASIC, which use a single window for programs, Amiga BASIC uses two: the Output window and the List window. When you first start Amiga BASIC, the List window is active and is in front of the Output window. The List window is where you write and edit programs; the Output window shows the results of a program. For example, a line-by-line listing of a program to calculate a monthly budget appears in the List window, while the budget itself — the results of the program — appears in the Output window. You can write individual programming statements in the Output window, but each one is run (executed) as soon as you type it.

Getting from one window to the other depends on what you're trying to do. Running a program shown in the List window automatically puts the results in the Output window. To get back to the List window, you can: click (the left mouse button) in the List window to make that window active; type List and press Enter; select Show List from the menu; or press Amiga key-L.

If a program to be shown in the Output window calls for some cursor movement or mouse actions, the pointer must be in the Output window before the program starts. Put the pointer in the Output window and click the left mouse button. Then start the program. Starting a program occurs in one of three ways: with the pointer in the Output window, type Run and press Enter; press Amiga key-R; or select Start from the Run menu.

ADVANCED FEATURES OF AMIGA BASIC

Line numbers. You can write Amiga BASIC programs with or without line numbers or labels. Programs are run in the physical order of the statements, that is, how they appear on the screen. If you do add line numbers or labels, the program can reference individual lines and jump to them if need be. To provide you the greatest programming flexibility, Amiga BASIC programs can be mixtures of numbered lines, labeled lines, and lines with neither numbers nor labels.

Variables. A BASIC variable holds a value, text or numeric, used by commands in a program. Giving each variable a name tells the program to substitute the variable's value in place of the name each time it occurs. Other programming statements can then substitute sequential values at the name depending on what you want the program to do. For example, the variable name could represent the addresses on a mailing list, and the program might be to print mailing labels for each address. Each time the program comes to the variable name, it would substitute an address for the name and print the address.

Text variables, such as in the addresses, are also called *string* variables. String variables can include numerals, as a ZIP code would, but they are not "computation" numbers. Names of string variables in a program must end in a dollar sign. For instance, the name of the address variable in the earlier example could be something like ADDRESS$ or MAILDATA$.

Numeric variable names can end in different symbols to indicate the type of variable. The symbols and their meanings are: % — short integer, & — long integer, ! — single-precision number, and # — double-precision number. *Single-precision* and *double-precision* refers to the amount of memory allocated to the numbers and the accuracy of computations performed with the numbers. Single-precision numbers are stored in only 4 bytes of memory, while double-precision numbers are stored in 8 bytes. Thus, naming a variable as single-precision saves memory space but at the cost of less accuracy when the variable value is used in a computation.

Typical numeric variable names are BALANCE%, SUM!, and TOTAL#. Variable names, both numeric and string, should generally be descriptive so you know what they refer to in the program. Some programmers, however, simply refer to variables with a single character such as X% or Y$.

Amiga BASIC also has a set of specific commands for naming variables without using the special symbols at their end. You type the command and then the name. The commands are DEFINT, for a short integer; DEFLNG, for a long integer; DEFSNG, for a single-precision variable; DEFDBL, for a double-precision variable; and DEFSTR, for a string variable.

Arrays. An array is similar to a variable, except that an array is a group of variables. Each variable in an array is known as an element. Like naming a variable, you also name arrays with a single name. The benefit of arrays is that you can reference multiple variables with that single name.

Arrays can have multiple dimensions. A one-dimensional array is a list of values; a two-dimensional array is a table having both columns and rows; a three-dimensional array would appear as a "cube" of values; a four-dimensional array is not easily visualized, but has the same fundamental properties as the others. In Amiga BASIC, arrays can have up to 255 dimensions.

Two Amiga BASIC commands specifically for arrays are DIM and SHARED. Use the DIM command for naming the array, setting its dimension, and defining the number of elements in each dimension. If your program consists of subprograms, the DIM command applies only to the specific subprogram in which the command appears. To create arrays that can be used by all the other subprograms and the main program, you use the SHARED command.

A typical use of an array is to assign values to other Amiga BASIC commands that require them. For example, the SAYcommand, which turns the Amiga into a speech synthesizer, requires nine elements to fully describe the sound. Individual elements control the speech volume, rate of speaking, inflection, and pitch, as well as other features. By listing your choices for the SAY's nine elements in an array, you determine how the computer's voice sounds when it speaks. The SAY command is described in more detail later.

Other Amiga BASIC commands for more advanced array processing are LBOUND, UBOUND, and OPTION BASE. Refer to the Amiga BASIC manual for their descriptions.

MULTITASKING

Multitasking is the process that, as its name states, lets your computer work on more than one task or program at a time. The tasks can be such things as reading a disk, moving a screen object, playing music, or printing a form. The programs can be any programs running simultaneously in individual screens. Even if a screen is completely obscured by other screens, its program can still be in progress.

The Amiga runs multiple tasks automatically, without any specific instructions from you. Opening a window and loading a program into it is all that it takes to have multitasking. The only significant noticeable limitations of multitasking are that some programs will run a bit slower when the Amiga is juggling a large number of tasks, and eventually memory will fill.

From your point of view the monitor may seem cluttered with a lot of screens, making them hard to keep track of or distinguish, but the Amiga has no trouble keeping them all in order. In fact, because of the way multitasking works, each task and program seems to have the entire machine to itself. The secret to Amiga's multitasking resides in its custom chips and ROM programs. Inside the Amiga, the 68000 chip works on the principle of interrupts. That is, the 68000 continues working on a task or program until it is interrupted by some other task or program. The interrupts are controlled by the custom chips, in particular the peripherals/sound chip that contains the interrupt controller circuits. Then, programs in ROM take the interrupt requests, assign them a priority, allocate memory space for each one, and coordinate other essential functions. When the 68000 is finally interrupted, it then begins to process the interrupt's job as if it were the only one in the computer, while the custom chips take care of the rest of the necessary work.

Instead of using interrupting, other computers operate by polling, which makes the CPU chip periodically survey the rest of the computer to see if any task other than the one in progress needs attention. Not only does this tie up the CPU chip, it restricts the computer to dealing with one job at a time. In other words, instead of carrying out a task until interrupted, the CPU in that computer runs a few steps of a task, stops, polls the rest of the computer, restarts where it stopped, and continues.

Multitasking and programming in Amiga BASIC can create a few interesting situations. For instance, the ON statements, such as ON MENU, ON MOUSE, and ON TIMER, automatically create interrupts for their systems. As an example, ON MOUSE tells a program to run a specified routine whenever the mouse's left button is pushed. An ON TIMER command similarly tells the program to run a routine after a certain period of time elapses. In both cases, the programming command is interrupting the 68000 to force it to run a routine, which, as some of the programs later in this book demonstrate, is the way to get the mouse to draw on the screen and select menu items.

Sometimes programming with interrupts requires that you understand what takes place during the interrupt procedure. An interrupt service routine might clear all registers and set memory locations to zero during interrupt processing. Trying to maintain a value in a register would then need some additional programming steps to save it. Also, an interrupt command may require that you keep close track of shared system resources, such as the printer or the ports. Imagine the problems if one program sent a few characters to the printer, then the next program did, and a third, and so on.

By and large, Amiga BASIC and multitasking get along very well. The programs in the rest of this book illustrate how you can interrupt the Amiga to get it to do what you want, when you want. And it happens very fast.

Part II

GRAPHICS ON THE AMIGA

CHAPTER 4

Understanding Graphics

The Amiga's graphics capabilities offer perhaps the most choices and options for programming creatively on your computer. If you've never programmed graphics before, the necessary concepts and terminology take some getting used to. If, on the other hand, you have already created some computer graphics (even on another microcomputer), you'll find some new tricks that make the job easier and more effective. Either way, with a little effort you can give vent to the artist — both within you, and the Amiga.

Because of the Amiga's design, you have three options for producing computer artwork: programming in machine language to access the hardware directly; programming using the special graphics routines in the Amiga's ROM libraries; and programming using a higher-level language such as BASIC. The programs in this book are in BASIC.

You can, of course, also create graphics with commercially available graphics programs that relieve you of the programming tasks. Those programs are usually written in machine language to take full advantage of the Amiga's speed and flexibility.

SOME NECESSARY TERMINOLOGY

Some of the terminology of computer graphics is as familiar as the labels on the controls of your TV set, while other terms require some understanding of how computers operate.

Graphics

Computer graphics means pictures that are dots, lines, and colors electronically drawn on the screen. Unlike the video pictures that you get from TV reception or a VCR, computer graphics must be drawn individually. A stationary graphic image on the screen can be a single drawn picture; to achieve motion or animation, you create multiple images and display them sequentially.

Currently, rudimentary equipment can digitize still images and input them into the Amiga as graphics. For instance, pointing a specially equipped camera at an object displays its image on the Amiga and stores it in memory as a digital image. Using the Amiga's graphics capabilities, you can then edit the image, manipulate its features, or add other graphics to it. Depending on the amount of free memory in the Amiga, you can input a number of digitized images, and by displaying them rapidly one after the other, create brief sequences of animation. Getting the images in color may require shooting the original object through three filters.

The term *graphics* also applies to printers that can produce pictures, such as the dots of a dot-matrix printer replicating the dots on the monitor. Most dot-matrix printers also have an "extended character set" which includes block and line characters for creating printed graphics. Printer graphics are similar in concept, but do not use the same BASIC commands as the Amiga's graphics.

In many microcomputers, graphics are differentiated from "text," or typed characters on the screen, but the Amiga does not have a special "text-only" mode. Text is handled as if it were graphics, which is why you can display different typefaces and character sizes on the screen.

Pixel

A screen consists of hundreds of dots which can be programmed to be on or off, light or dark, and different colors. It is the patterns produced by the dots that create an image on the screen. Each dot is a picture element, or *pixel* (Figure 4.1).

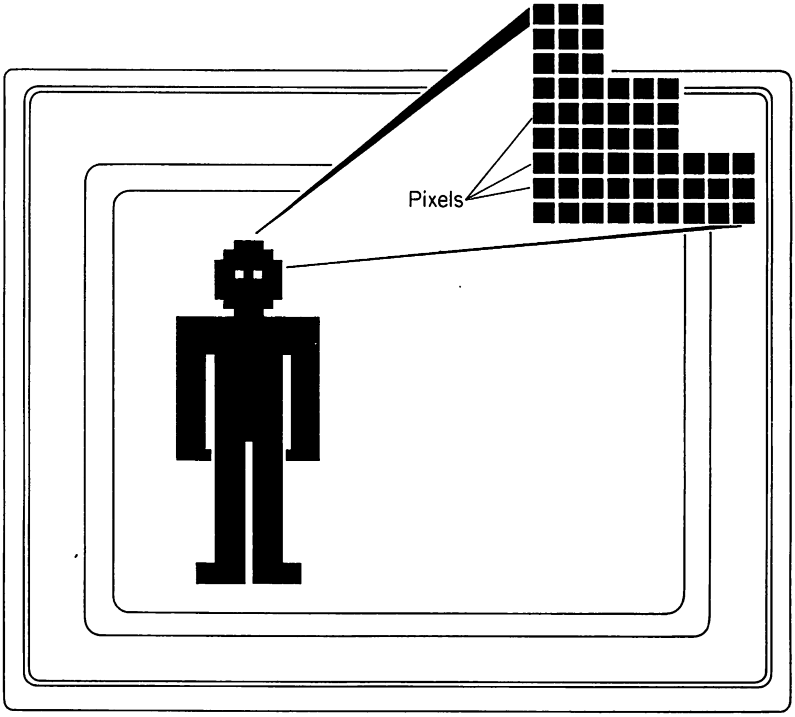


Figure 4-1

Pixels are arranged on closely spaced horizontal lines (scan lines) on the screen. Inside a color TV or monitor three electron guns sweep, or scan, together across the screen at each line (a black-and-white TV or monochrome monitor has only a single electron gun). As they scan, the three color beams' intensities are modulated at each dot on the line, producing a pattern of colored pixels. For example, at one pixel the red gun can be on its highest intensity, but the green and blue guns at their lowest intensities to produce a pure red pixel. At the next pixel a different mixture of gun intensities produces a different color for that pixel.

A pixel remains "lit" on the screen after the beam leaves it because the phosphor coatings (one for each color) on the inside of the tube retain images for a short period of time. (It's this fact that causes an afterimage to remain briefly on your monitor after you've turned off the power.) But the phosphors lose the image quickly, so the beam must rescan the screen to "refresh" the image. Normally the refresh rate is 60 complete screen scans per second. And if the beam is modulated at a pixel differently than it was during the previous scan, the pixel takes on a different color and the screen shows a different image.

The modulation of the beams in a TV set come from the TV signals received by the antenna; for a computer picture, the modulations come from the computer memory. Pixels get their color assignments from bit values at specific locations in the Amiga memory. (See bit map later.) The term *pixel* is also used as the measurement for describing the resolution of the screen mode. For example, one Amiga graphics mode is 320 pixels per line on the screen, while another mode is 640 pixels per line.

Pixels are sometimes referred to as being either horizontal or vertical. This has nothing to do with their shape. A pixel is simply a dot. The number of horizontal or vertical pixels is a way of indicating the screen size — how many pixels across and down. For instance, these references to a screen's pixels all mean the same thing:

* 320 horizontal pixels by 200 vertical pixels

* 320 horizontal pixels by 200 rows

* 320 *pels* by 200 rows

"Rows" does not refer to the rows of characters that you can type on the screen, but instead are the number of screen scan lines. The abbreviation pel (for picture element) is sometimes used in place of pixel.

Pixels also correspond directly to the x- and y-coordinates of a point on the screen. That is, many screen graphics use the Cartesian coordinate system to draw points, lines, and figures. The horizontal (x) value refers to the number of pixels in the horizontal direction, while the vertical (y) value refers to the number of pixels up or down the screen.

Bit Map and Bit Plane

A bit map is an area of Amiga memory that determines the colors of pixels on the screen. The bit map is divided into subareas known as bit planes (Figure 4.2), and each bit plane assigns 1 bit to a pixel. Combining

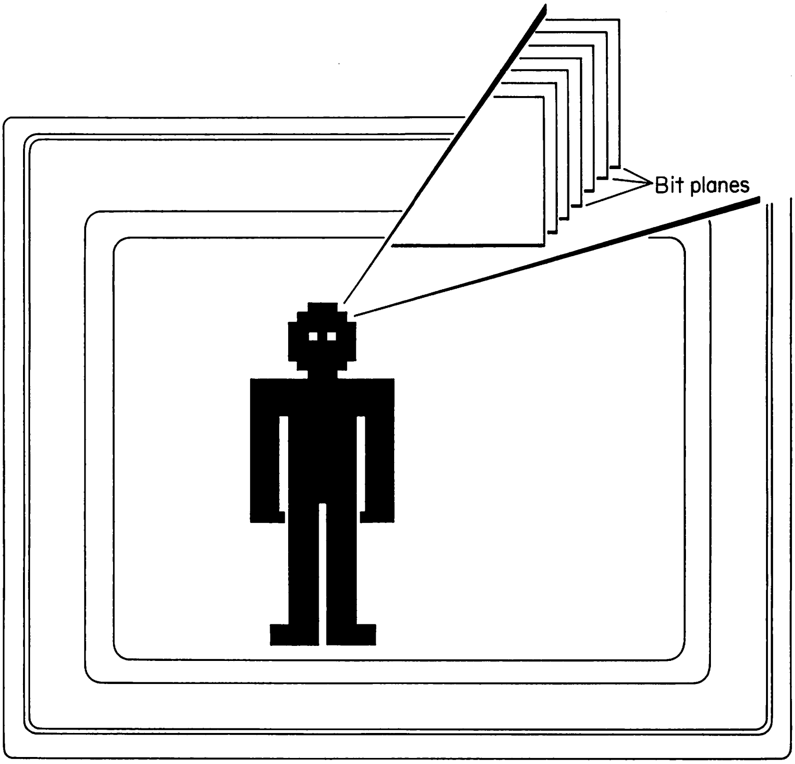


Figure 4-2

the bits from each of the bit planes produces the bit pattern that describes the color of the pixel. The Amiga then matches the bit pattern with a color register table and sends the appropriate signals to the monitor's electron guns.

You can think of bit planes as separate "sheets of bits" that produce the patterns for each pixel. When you create an image using multiple bit planes, the image is called a *raster*.

Programming in BASIC does not require you to create individual bit planes and bit maps for an image. The various graphics commands do it for you automatically. But certain of the commands offer variable inputs that assume you know the fundamental concept of bit planes.

Colors, Color Register Table, and Palette

The bit patterns sent to the monitor's electron guns can modulate intensities to 16 different levels. Because a color monitor or TV creates colors using three electron guns, the Amiga can produce 16^3 (16 to the third power), or 4,096 colors. In most of the Amiga's resolution modes (see resolution and modes later), the screen can display up to 32 separate colors at a time. This requires using 5 bit planes to describe the bit patterns for each color. In other words, 5 bit planes translate into a bit pattern 5 bits long. The 5 bits, in turn, can be ordered into 32 separate arrangements to correspond to 32 colors available to the screen at any one time. The colors correspond to color assignments in a color register table.

If you define a lesser number of bit planes for the colors, the screen is limited to the maximum number of bit arrangements. For example, if you define only 2 planes, the maximum number of colors is 4, i.e., 00, 01, 10, and 11. Those 4 bit arrangements then match 4 colors in a color register table. Having the colors listed in a color register table means that the Amiga's colors are a software function and thus are not "hardwired." You can change colors by changing the reference to the color register table; the actual bit pattern, however, can stay the same.

Because the color register table lists all the colors available for an image, it constitutes the *palette* for that image. To an artist, a palette is a board on which to mix paints and pigments. To a computer, a palette is a range of colors that can be assigned to an image. The Amiga BASIC command PALETTE is for defining a set of up to 32 colors. But you can define the full 32 colors for a scene only if you have first set the screen (with the SCREEN command) to have 5 bit planes. That is, it takes 5 bits to achieve 32 separate bit patterns; therefore, you need 5 bit planes to get the five bits per color.

The Amiga BASIC manual refers to each of the colors in a palette as a "paint can," and has settings for red, green, and blue for each color. It is these settings that modify the intensities of the three guns in the TV or monitor.

Resolution and Mode

Resolution is a measure of image clarity and sharpness on the monitor or TV screen. RGB monitors have the capacity for high resolution. TV sets are usually limited to low resolution. Using the BASIC SCREEN command, you can set the resolution for graphics to correspond to the type of screen on your Amiga. These resolution settings are called the screen *modes*. The high- and low-resolution settings for the screen modes refer to the number of pixels per horizontal line; that number is also called the *pixel width* of the screen. High resolution is 640 pixels per line; low resolution is 320 pixels per line. The pixels in low resolution are thus twice as wide as those in high resolution.

The screen modes also include settings for *interlacing*, which determines the number of lines on the screen. Normally, a TV screen has 200 lines "stacked" vertically on it. That is, the beam sweeps back and forth across the screen 200 times to create a complete image. The 200 lines are "noninterlaced," which means they are their normal distance apart. Interlacing stacks 400 lines on the screen. First, the beam writes all 200 odd rows (1, 3, 5, . . . 399), then it goes all the way back up to the top (during a period called "vertical retrace") and writes all 200 even rows (0, 2, 4 . . . 398). Essentially, at each sweep the beam moves down the screen only one-half the normal distance to begin the next scan. The extra lines are "interspersed," or interlaced, between the normal scan lines.

Interlacing doubles the vertical resolution and is thus most appropriate for RGB monitors. However, interlacing has its price. Because the beam must scan twice as many lines on the screen, it refreshes the image only 30 times a second instead of 60 times. This creates a screen flicker which, depending on the colors of the image, can seriously affect viewing. Flickering is most noticeable on high-contrast images.

Interlacing and resolution affects Amiga text as well as graphics (which is not surprising since Amiga treats text characters as graphics). For example, the default text characters (known as Topaz 8) fit 80 to the line when the resolution is set to high, but only 40 to the line at low resolution. Similarly, in a noninterlaced mode the screen shows 25 lines of Topaz 8 text, but 50 lines in interlaced mode.

The Amiga's resolution also affects how many colors you can assign to a screen. Low-resolution screens can show up to 32 colors, but high-resolution screens are limited to 16 colors at one time.

Contrast

Contrast is the degree of difference between an image's light and dark sections. High-contrast images have very dark and very light sections — blacks are truly black and colors are deep and rich. Low-contrast images appear to blend together, with blacks tending to be gray and colors less intense. Generally, the higher the contrast, the more pleasing an image is to the eye.

Contrast is controlled by a separate dial on your computer's monitor. Monitors made strictly for computers usually have high-contrast screens; TV sets normally do not.

Foreground and Background

When a character is written on the screen, the character itself constitutes the foreground and the rest of the screen is the background. On a black-and-white screen where the letters are white, the foreground color is white and the background color is black. You can set the Amiga to numerous foreground and background color combinations using the BASIC statement COLOR.

Coordinates

Coordinates are a set of two numbers that identify the location of a pixel (or point, dot, or pel) on the screen. The two numbers of the set are referred to as (x,y), where *x* is the horizontal position of a pixel and *y* is the vertical position. On the Amiga screen, the upper left corner is considered position (0,0); the position one pixel to the right is (1,0); and a position one pixel directly below that is at (1,1). If you wanted the computer to draw a point of light on the screen, you could identify where the point is to go by specifying its coordinates.

Another way of thinking about coordinates is that the first number in the set is for moving a point over to the right and the second number is for moving the point down. Thus, instead of saying (x,y) or horizontal and vertical, the two numbers are "over and down."

Coordinate numbers are relative to the resolution mode of the Amiga's screen. For example, if the screen is at low resolution and noninterlaced (which sets the screen to have 320 pixels across and 200 pixels down), the center of the screen is at coordinates (159,99) and its lower right corner is at (319,199). A screen at high resolution (which is set for 640 pixels across and 200 of them down) has its center at coordinates (319,99) and lower right corner at (639,199). (See Figure 4.3.)

Coordinates are also influenced by the features you assign to a

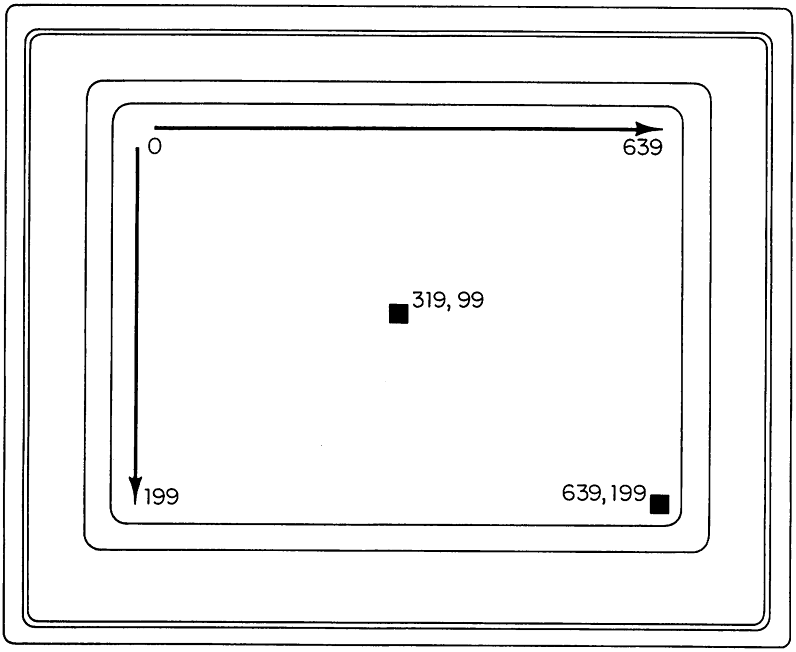


Figure 4-3

window (see the later description of windows). For example, if a window on a high-resolution, noninterlaced screen includes a border and room for a sizing gadget, the maximum x-coordinate is 617 instead of 639 and the maximum y-coordinate is 186 instead of 199. The border automatically takes 8 pixels from the display, and the sizing gadget takes 14 pixels. Thus, for the example, the upper left corner of the display is still (0,0), but the lower right corner is (617,186). In low-resolution mode, the number of pixels assigned to the border and sizing gadget are the same as in high resolution.

Why are coordinates important? You use them to designate positions for objects on the screen as well as to draw points, lines, and figures. The Amiga BASIC statement PSET (x,y), for example, draws a point of light on the screen described by the numbers you substitute for the *x* and *y*. Also, animated figures are moved relative to a set of coordinates.

Screens and Windows

In normal, everyday usage the term *screen* refers to the monitor's visual display, but when describing the operations of the Amiga, *screen* has a somewhat different meaning. An Amiga screen defines the traits that the visual display can have. You define those traits with the BASIC SCREEN command. They include resolution and interlacing, display width and height, and number of bit planes. Each screen also has an identifying number. Screens must be as wide as the monitor display, but can be shorter or taller than the display. Screens less than the display height must appear only at the bottom of the display.

The Workbench screen that appears when you run the Workbench disk is an example of an Amiga screen. Its settings are high-resolution, noninterlaced mode; full height and width of the display; and two bit planes.

Having defined the features of a screen with the SCREEN command, you can then define *windows* that fit within the screen. Those windows will have the features of the screen — i.e., a high-resolution screen can only show high-resolution windows — but windows also have their own traits. You define them with the BASIC statement WINDOW.

A window is where results (or output) from your programs will appear. Just as the Workbench screen can show numerous windows, a custom-defined screen can also hold many windows. You must define each one with a separate WINDOW statement. As described in the Amiga BASIC manual, the WINDOW statement assigns a number, title, size, type, and screen to each window you define. Other forms of the WINDOW command open and close a window, and report on its status.

Windows can have the drag bars, sizing gadgets, close gadgets, and the other standard features seen on Amiga windows. Windows can also be a different size from the full screen.

DEFINING A GRAPHICS DISPLAY

If the Workbench screen or the BASIC Output screen has the features sufficient for graphics, you can use either of them as the "artwork canvas." A customized screen with its own unique features, on the other hand, requires that it be defined first. You define a graphics display screen with three Amiga BASIC statements: SCREEN, WINDOW, and PALETTE.

The SCREEN Statement

The SCREEN statement has two forms. One form opens a screen and defines five features for its display. The five features are screen number, width, height, depth, and mode. The other form of the SCREEN statement closes a screen.

The first item to be defined is screen number, a number from 1 to 4 that is simply an identifier for that screen. You can have up to four screens defined at one time. Width is the width in pixels for the screen on the display. For a low-resolution screen, a width of 320 fills the display; a width of 640 fills the high-resolution screen. Height is the vertical height in pixels for the screen; a noninterlaced screen with a height of 200 fills the display, while a height of 400 fills the display for an interlaced screen setting. Depth refers to the number of bit planes, and therefore the number of colors, assigned to the screen. Depth is a number from 1 to 5. A depth of 1 yields two colors, 2 yields four colors, 3 yields eight, 4 yields sixteen, and a depth of 5 yields thirty-two colors. Mode sets the resolution and interlacing. Mode can be a number from 1 to 4. Mode 1 is low resolution (low-res), not interlaced; mode 2 is high resolution (high-res), not interlaced; mode 3 is low-res, interlaced; and mode 4 is high-res, interlaced.

An example SCREEN statement to set the screen at high-res, not interlaced, full screen, with ability to show sixteen colors is:

```
SCREEN 1,640,200,4,2
```

The form of the SCREEN statement to close a screen is SCREEN CLOSE followed by a screen number. The first number in the SCREEN statement is the screen number. Thus, the statement SCREEN CLOSE 1 closes the screen defined in the previous example.

The WINDOW Statement

Like the SCREEN statement, one form of the WINDOW statement describes the display features of a window and then opens it. The WINDOW statement has three other forms as well.

The form of the WINDOW statement to describe features has five inputs: window ID number, a title, rectangle, type, and screen number. The window ID number is an identifier number for that window. Because you can have numerous windows on one screen, the number can be from 2 to *n*. (Window 1 is already defined as the BASIC Output window.) A window title, which will appear at the top of the window, is optional. Rectangle refers to the size and position of the window. You specify the rectangle by listing the coordinates of the top left and bottom right corners of the window. Leaving this item blank makes the window cover the entire screen. Type describes the gadgets the window will have, and whether graphics on the window are saved when it isn't the active window. The numerical settings for type are:

Setting	**The Window will have:**
1	Sizing gadget
2	Drag bar
4	Back gadget
8	Close gadget
16	Enough memory to remember its graphics when not active

For the window to have more than one of the gadgets, add the setting numbers. For instance, for a window to have the Drag bar and Back gadget, add 2 and 4 for a type value of 6.

The last item in describing a window is screen number. This refers to the screen you described earlier that will hold this window. Screen number can be from 1 to 4.

An example statement to give a window the title of "BALLGAME," fill the entire screen, have a Size gadget, Drag bar, Back gadget, Close gadget, and enough memory, and be assigned to the screen defined earlier is:

```
WINDOW 2,"BALLGAME",,31,1
```

Note that the rectangle input is null (,,) so the window will fill the entire screen. The WINDOW statement automatically opens a window with the defined features.

The form of the WINDOW statement, WINDOW CLOSE, closes a window (makes the window invisible). The window number following the statement closes that window. To close window number 2, the statement is WINDOW CLOSE 2. A WINDOW statement followed by the window number (and no other inputs) reopens a closed window. WINDOW 2 reopens the closed window number 2.

WINDOW OUTPUT followed by the window number makes the window active but does not put it in front of any other windows. In other words, WINDOW OUTPUT sends graphics or text to a covered window without affecting the visible window.

The fourth and final form of the WINDOW statement reports the status of a window. As described in the Amiga BASIC manual, WINDOW(*n*), where *n* is a number from 0 to 8, reports different items of status. The WINDOW(8) statement is particularly important for programs that access the Amiga's library subroutines (see later for a description of these routines). WINDOW(8) sends output from the library routines to the current Output window.

The PALETTE Statement

The PALETTE statement works in conjunction with the COLOR and SCREEN statements to define the colors of up to thirty-two "pens" on a window. PALETTE requires four inputs: pen number, and values for red, green, and blue. The pen number is the numerical "name" of a particular color in your palette. For instance, if in the SCREEN statement you defined a depth of 4, you can define sixteen separate colors, or pens, with the PALETTE statement.

The values for red, green, and blue range from 0 to 1 in decimal hundredths and control the relative intensities for each electron gun of the monitor. That is, the Amiga sends the values to the circuitry of the monitor tube which then modulates the voltage levels of the individual guns to produce brighter or dimmer intensities. For example, a color with the setting of 0 for red, 0 for green, and 1 for blue produces a pure blue color because the red and green guns are at minimum intensity. A setting of 1 for red, .13 for green, and .93 for blue mixes the three colors and produces a color of violet. Similarly, .9 for red, .7 for green, and 0 for blue produces gold.

Suppose you want colors 11 and 14 to be blue and gold, respectively. The PALETTE statements would be:

```
PALETTE 11,0,0,1
PALETTE 14,.9,.7,0
```

A PALETTE statement for each color describes it for a pen. Thus, if you have defined a depth of 4 for a screen, you must define sixteen PALETTE statements for the sixteen possible colors. The colors do not have to be similar; they can be any ones that you want.

Changing the color of a pen instantly changes the color of every pixel on the screen drawn with that pen. This is one way to do limited, but very fast, animation.

After describing all the colors with the PALETTE statements, you then use the COLOR statement to assign colors to the pens for the screen's foreground and background colors. The foreground is the colors of letters or drawn figures; the background is the color of the rest of the screen. For instance, if color number 11 is blue and 14 is gold, to draw a blue line on a gold background, the COLOR statement would read:

```
COLOR 11,14
```

To change the color of only the foreground pen, you simply leave the number for the background pen blank.

Once the screen and its colors are completely defined, you're ready to begin drawing. You can draw points, lines, and shapes and fill them with color. And by linking successive images together with special Amiga BASIC commands, you can animate your graphics. Chapter 5 describes the fundamentals of drawing graphics, and chapter 6 explains the process of computer animation.

SOME ADDITIONAL GRAPHICS FEATURES

Although Amiga BASIC provides a full set of commands and statements for drawing and animating graphics, other commands are available to complement BASIC. These commands are in the Amiga library routines stored in a file named "Graphics.bmap" which is already on the BASIC disk in the BasicDemos directory. The library includes programming subroutines that help draw polygons, identify the color of a particular point on the screen, move the cursor without drawing a line, and create lines with patterns.

You can access these additional commands with the Amiga BASIC commands LIBRARY, DECLARE FUNCTION, and CALL. The command LIBRARY "graphics.library" tells the Amiga to access subroutines in the library named "graphics.library" which is in the "Graphics.bmap" file. The DECLARE FUNCTION command then names the specific subroutine in the library that you want to use. For example, the subroutine named Move( ) moves the graphics pen, while the subroutine named Text( ) draws text characters on a graphics screen.

The CALL command starts the subroutine. For example, CALL Move&(RP&,x&,y&) is the format for calling the Move& subroutine. The & symbol makes the variables long integers and RP& item refers to the RastPort of the display — RP& = WINDOW (8) sends the output to the current Output screen. The symbols x& and y& refer to the x- and y-coordinates of the cursor. You can use this statement to position the cursor at a precise pixel on the screen.

Other subroutines change the "drawing mode" of the pen colors. The default mode (the one that normally works with the COLOR statement) is known in the library as JAM2. Another mode is JAM1 in which only the foreground pen draws in color. A mode known as COMPLEMENT changes the colors of both pens to their complements as described in the PALETTE table of colors. The fourth mode is called INVERSID and reverses the functions of the two pens.

By accessing built-in subroutines, these advanced graphics procedures are at a "lower level" of programming than BASIC. However, many of the subroutines are duplicated by BASIC commands, so the primary use of the low-level Amiga library is to achieve faster results and to give you more programming flexibility. For more information about the library of Amiga graphics functions, see the *Amiga ROM Kernal Manual.*

CHAPTER 5

Drawing with the Amiga

If you've seen any of the commercial graphics software programs in action, or have run the demonstration programs, you know how crisp and clear Amiga images are on the screen. Even without artistic training and talent, your pictures can rival the quality of Saturday morning cartoons. And with a little practice your Amiga graphics can be used for business presentations, personalized greeting cards (if you have a good printer), announcements, games, puzzles, or just to tack on the refrigerator door.

Simple commands in Amiga BASIC draw images on the screen. Specific commands draw points, lines, shapes, and patterns, as well as color-in closed figures and draw with colored "pens." A special animation creation system is for drawing objects that you want to animate with other programming statements. Chapter 6 describes the process for creating animation.

Drawing with the BASIC commands does not use the mouse to produce images, but instead uses coordinate references to position images and drawing pens. At the end of this chapter a program creates an Amiga BASIC drawing language for producing images with coordinate references. The language is essentially a "stripped down" version of turtle graphics made popular with the Logo language. To understand the commands in the language, you first need to know about Amiga BASIC's drawing statements which are described first.

THE AMIGA BASIC DRAWING COMMANDS

Drawing a Point

The simplest "image" on the screen is a single point. Two commands draw a point: PSET and PRESET. The PSET command draws a point with the foreground pen, while PRESET draws the point with the background pen.

To draw a point on the Amiga BASIC Output window (which is on the Workbench screen), just type one of the two commands followed by the coordinates for the point's position. For example, the default colors of the Workbench screen are foreground white and background blue. To draw a white point on the blue background at screen position (64,40), you type:

```
PSET (64,40)
```

The command PRESET (64,40) also draws a point at the same location, but because the pen color is the same as the background color, you don't see the point. So what's the use of PRESET? Primarily to erase points drawn in other colors. Thus, after drawing a white point with PSET (64,40), the PRESET (64,40) erases it. Actually it doesn't erase it in the sense of a rubber erasure, rather it erases the point by overwriting it in the same color as the background. (Amiga BASIC does have an ERASE command, but it has nothing to do with erasing graphics. The ERASE command erases arrays.)

Both PSET and PRESET have options for positioning the point relative to the last one that was drawn. In the preceding examples, the coordinates designate a position on the screen regardless of the previous position of the cursor, but by adding the STEP option, you can move the point relative to the previous point. For instance, continuing with the previous example, the command PSET STEP (20, − 30) draws a new white point at the absolute screen position of (84,10); the x-coordinate adds to the previous x-coordinate — 64 + 20 = 84 — and the y-coordinates similarly are combined — 40 − 30 = 10. Note that positive numbers move the position to the right or down, while negative numbers move the position to the left or up.

Typically, the STEP option is for drawing new images some specific distance from the previous image. You don't have to figure out the exact screen coordinates for the beginning of the image. The STEP option also saves a lot of time when changing image locations. Changing the starting point coordinates changes all the other images' positions relative to that initial point.

Many other Amiga BASIC drawing commands use the STEP option to make their respective functions relative to a previous screen position. One trick in using the STEP option with PSET is to put points along vertical and horizontal grid lines to provide an on-screen reference system. Then, as you begin to draw images, they can be relative to a specific (x,y) reference point.

To draw points in different colors, change the pen colors with the COLOR command before the PSET or PRESET commands. For example, COLOR 3,4 changes the pens to colors 3 and 4 as defined by the PALETTE statements. If you defined a customized screen and set of colors, you can change the pens to any of them and draw different colored points on the screen.

As you're drawing images, it's easy to lose track of a particular point or area's color, especially when the screen shows sixteen or more colors and the image includes many shades of similar colors. To find the color of a particular point on the screen, use the POINT command and the point's (x,y) coordinates. The command POINT (64,40) will display a number on the screen corresponding to the point's color number as designated by PALETTE statements. For instance, on the Workbench screen, white is color number 1. After drawing a white point with PSET (64,40), POINT (64,40) returns the number 1.

The command for clearing all the points (and other images, too) from the screen is CLS. Before typing CLS, make sure to save the image or it is lost for good.

Single points are so small you may have to look hard to find one. In fact, a single point as a "graphic" is not too visually inspiring. Lines and shapes most likely will be of more use.

Drawing Lines and Shapes

Amiga BASIC draws lines by connecting the coordinates of the two endpoints. The coordinates can be absolute or, using the STEP option, can be relative to the last point of the previous line. Shapes are drawn by simply connecting lines. Specific commands draw a box and a circle; to draw polygons, you connect the lines with successive line-drawing commands.

The LINE command draws a line. To draw a line from the point (20,30) to the point at (160, 100), type the statement:

```
LINE (20,30)-(160,100)
```

The line is the color of the foreground pen. To draw the line in another color, put the pen number after the coordinates. For instance, if you had defined a set of PALETTE statements so pen number 6 is red, the statement

```
LINE (20,30)-(160,100),6
```

draws a red line.

Boxes

Drawing a box is not much more difficult. Adding the letter *b* after the pen color number makes the Amiga draw a box. The coordinates of the two points identified in the line statement are the box's opposite corners. Thus, to draw a red box (assuming your PALETTE statement defines pen number 6 as red), you could type:

```
LINE (20,30)-(160,100),6,b
```

If you want the box to be in the default foreground pen color, do not put the 6 in the statement. For instance,

```
LINE (20,30)-(160,100),,b
```

does the trick. The pen number place must be included but is null.

Change coordinates to draw other boxes on the screen. For instance, this statement draws a box inside the first one:

```
LINE (30,40)-(140, 80),,b
```

To fill the box with the screen's foreground color, add an *f* after the *b*, as in this example:

```
LINE (20,30)-(160,100),,bf
```

The *bf* stands for box filled, not box foreground. To fill the box with other colors, type a different pen number color between the two commas. So, to fill the box with red, type:

```
LINE (20,30)-(160,100),6,bf
```

If you're following along with these statements, typing them on the screen, you'll notice that previous images get in the way of your new images. Remember to type CLS before starting another image. The "fill with color" capability for a box is replicated by other Amiga BASIC commands. They are described later.

More on the STEP Option for LINE

The LINE command has the STEP option for drawing lines relative to the last point referenced by a previous BASIC command. To connect lines, start the new line at either endpoint of the previous line. The STEP statement can be in front of either set of coordinates. The statement LINE (20,30)-STEP (140,70) draws the same line as LINE (20,30)-(160,100) because the STEP option adds the coordinates to the previous ones in the LINE statement. That is, STEP used in front of the second set of coordinates makes them relative to the first set.

Putting the STEP option in front of the first set of coordinates makes them relative to the last point that was referenced by another LINE (or other) statement. For example, the following two statements draw two connected lines (Figure 5.1):

```
LINE (20,30)-(160,100)
LINE STEP (0,0)-(180,120)
```

Continuing to draw lines in this fashion until they meet — the last one would end at the starting point of (20,30) — results in a shape that is a closed polygon. However, the AREA command (see later) is another way of drawing a closed figure.

The STEP option used with the box option draws a square. For instance, the statement

```
LINE (20,30)-STEP (60,60),,b
```

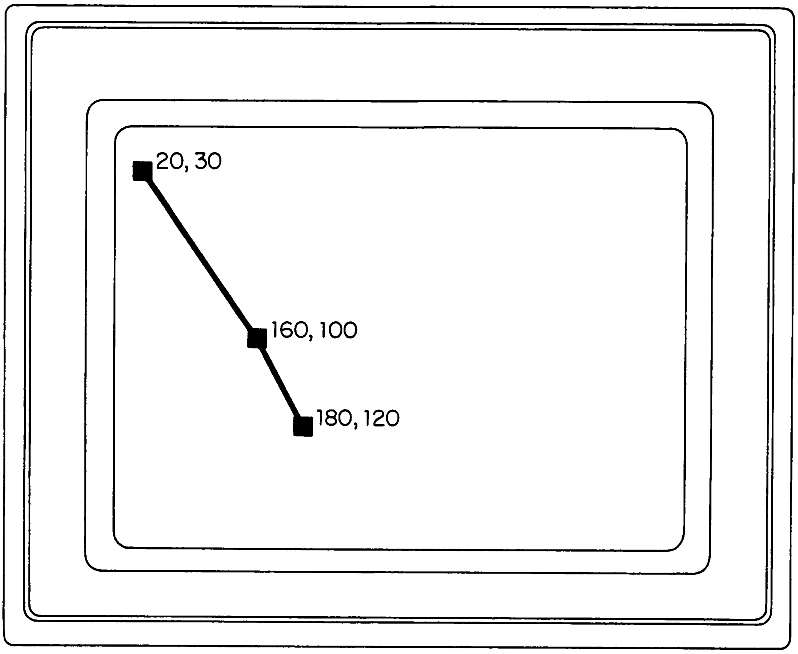


Figure 5-1

draws a box 60 pixels to the side, with its upper left corner at (20,30). The drawing starts at (20,30) and then "STEPs (60,60)" pixels for each side. (Amiga BASIC has two other LINE commands: LINE INPUT and LINE INPUT#, but they are not for drawing lines or other graphics. Those two commands read a line of characters that you type on the keyboard.)

Circles and Ellipses

The BASIC statement CIRCLE draws both circles and ellipses. It works in about the same way as the LINE statement except the inputs are the coordinates of the center of the circle and the length in pixels of its radius.

A typical CIRCLE statement is:

```
CIRCLE (80,100),50
```

The (80,100) identifies the circle's center point, and the 50 is the length of the radius.

To color the circle, add a pen number after the radius number. So, for example, to draw a red circle, the statement would be

```
CIRCLE (80,100),50,6
```

where the 6 identifies red from the Amiga's table of pen numbers.

Two other options on the CIRCLE command are starting and ending points if you only want part of a circle. By specifying the starting and ending points in radians, the circle can simply be an arc or a wedge such as you might use for a business pie chart. Radians are measured in units of pi, ranging from −2 pi to 2 pi. If the radians are positive, the circle is unfinished and the image is an arc; if the radians are negative, a line drawn from the endpoints turns the circle into a wedge (Figure 5.2).

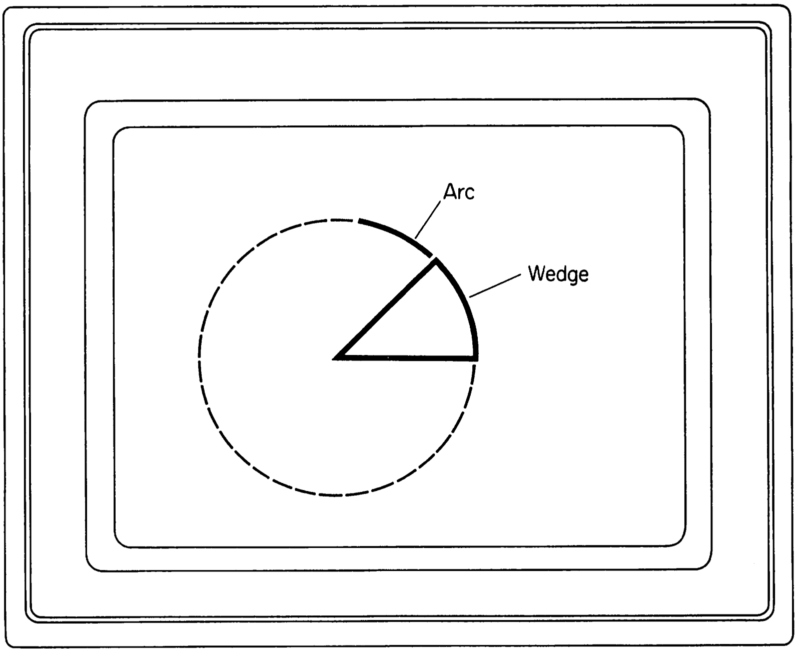


Figure 5-2

A statement to draw a circle with a 90-degree segment missing is:

```
PI=3.14159
CIRCLE (160,100),60,,-PI,-PI/2
```

The STEP option for the CIRCLE command pertains to the coordinates of the center point. CIRCLE STEP (40,50) draws a circle with its center offset 40 pixels in the x direction and 50 in the y direction from the last referenced point.

The Whole Point of STEP is...

...to keep you from going blind trying to count (x,y) coordinates for every image. The STEP option is relative to the last referenced point in a program, regardless of whether the reference was in a PSET, LINE, or CIRCLE statement. Thus, if the program has a LINE statement that ends with the point (40,50) and the next statement is CIRCLE STEP (20,30), the circle's center is at (60,80). In other words, STEP refers to the point referenced in the LINE statement. A STEP (0,0) keeps the cursor at the last referenced point. Using that fact, you can draw concentric circles without worrying about the center point of each one. Simply figure out the first circle's center and make the rest of them STEP (0,0). Then if you want to move all the circles later, changing only the point of the first one so it moves, makes the rest move, too. As you get used to STEP, you'll see it's the right step for programming easily.

Often a circle on a computer's screen looks more oblong than round. This is an individual characteristic of each screen. To make circles look completely round on your screen, the Amiga can draw the circle as a slight ellipse to correct the distortion. Adding a number to the end of the CIRCLE statement draws an ellipse.

A CIRCLE statement such as

```
CIRCLE (80,100),50,4,,,5/18
```

draws a horizontally elongated ellipse. The 5/18 is a number called the *aspect ratio* of the circle. Aspect ratio is the numerical relationship of the horizontal to vertical shape of the circle. If the aspect ratio is less than 1, the ellipse is elongated sideways; if it is greater than 1, the ellipse is elongated horizontally (Figure 5.3). The standard Amiga monitor set to high resolution has an aspect ratio of 0.44. Other monitors tend to be closer to 1. How close depends on your computer's screen. If the manufacturer of your monitor doesn't list the aspect ratio, you can only find out by trial and error.

The following brief program draws a target consisting of three concentric ellipses:

```
CLS
CIRCLE (80,100),50,6,,,5/18
CIRCLE STEP (0,0),80,6,,,5/18
CIRCLE STEP (0,0),110,6,,,5/18
RUN
```

Each of the ellipses has the same center and aspect ratio; only the

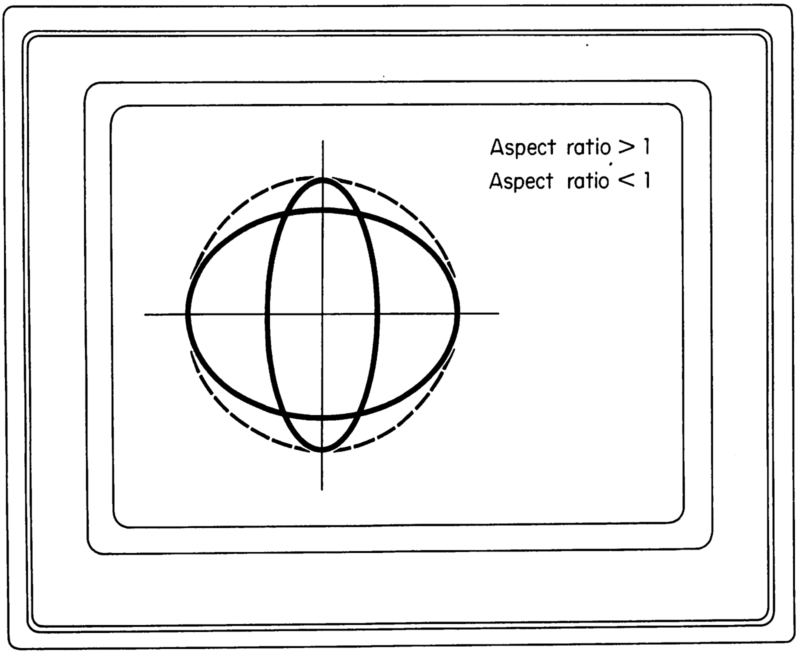


Figure 5-3

radius changes. Changing the aspect ratio can produce a set of circles with some apparent perspective and depth.

Aspect ratio also affects squares and other polygons on the screen. For the same reason that circles don't appear perfectly round, squares may appear rectangular, e.g., longer horizontally than vertically. Changing a square image's aspect ratio can make it look like a real square.

Drawing Closed Shapes Filled with Colors

Two Amiga BASIC statements — AREA and AREAFILL — are for drawing closed shapes and filling them with colors. The PAINT statement is similar and fills shapes drawn with the CIRCLE and LINE statements. The AREA statement connects up to twenty successive points. The number of AREA statements determines the number of sides on a polygon. Thus, the three statements

```
AREA (20,30)
AREA (40,50)
AREA (60,10)
```

draw a triangle with its vertices at the three referenced points.

The STEP option works with AREA as you might expect. The following statements draw the same triangle as in the previous example:

```
AREA (20,30)
AREA STEP (20,20)
AREA STEP (20,-40)
AREAFILL 0
```

The AREAFILL statement fills a closed polygon with patterns defined by the PATTERN statement. AREAFILL is placed following the AREA statements that draw the polygon. If you draw closed polygons with the LINE or CIRCLE statements, the PAINT statement can fill them with colors. Reference any point inside the polygon to tell the Amiga what to paint. For example,

```
PAINT (130,130),1,6
```

colors the circle drawn by CIRCLE (100,100),50 because point (130,130) is inside it. The numbers 1 and 6 refer to the colors for the polygon's interior and border, respectively. If the colors defined by the PALETTE statements define white as 1 and red as 6, the triangle has a white interior and a red border.

The STEP option offsets the (x,y) coordinates as it does with the LINE and CIRCLE statements.

Note that PAINT should only be used with figures you're certain are fully closed. Any breaks in the edges of the image give PAINT a chance to "leak" out. When this happens, the entire screen is painted with the pattern. Also, make sure the pen number you specify for the border color corresponds to the line that constitutes the polygon. If the numbers differ, the paint also leaks out.

The PATTERN Statement

The PATTERN statement creates lines and areas that have patterns of colors instead of solid colors. The form of the PATTERN statement for lines is PATTERN *n*, where *n* is a number representation of a bit pattern for the line. It works like this: *n* is a 16-bit expression that identifies whether the foreground or background pen will be drawing the line. When a bit is set to 1, the foreground pen will draw, and when the bit is 0, the background pen draws. Suppose you've set the colors for the foreground and background pens to be blue and gold, respectively, and you want to create a pattern of alternating blue and gold dots. The bit pattern would be 1010101010101010. At each 1, the blue pens draws a point, and at each 0, the gold pen draws its point. Or, suppose you want the pattern to be four blue then four gold dots on the line. That pattern would be 1111000011110000.

Once you've decided on the dot pattern, you then have to translate it into the number for the PATTERN statement. You've got two choices: figuring out the equivalent number for the binary bit pattern or translating it into hexadecimal. For example, the binary number 1010101010101010 translated into a normal number (base 10) is 43,690. If you want to use that system of determining pattern numbers, then the PATTERN statement for alternating dots would be PATTERN 43690. The hexadecimal system, however, is easier to use.

The hexadecimal system of counting has 16 digits — 0 through 9 and A,B,C,D,E,F — and can be represented by groups of 4 binary digits (because 2^4 is 16). Thus, the 16 bits needed for the pattern line can be represented by 4 hexadecimal numbers. Using the following table, you can determine easily the bit patterns and subsequent hexadecimal numbers for the PATTERN statement.

Hexadecimal	Bit Pattern	Pen Pattern (B = background F = foreground)
0	0000	BBBB
1	0001	BBBF
2	0010	BBFB
3	0011	BBFF
4	0100	BFBB
5	0101	BFBF
6	0110	BFFB
7	0111	BFFF
8	1000	FBBB
9	1001	FBBF
A	1010	FBFB
B	1011	FBFF
C	1100	FFBB
D	1101	FFBF
E	1110	FFFB
F	1111	FFFF

To draw a patterned line with alternating colors, starting with the background pen, find the pattern 0101, which is hexadecimal 5. Now the PATTERN statement becomes PATTERN &H5555. The &H tells the Amiga that the number is a hexadecimal integer. The four 5's describe the pattern. You always need four hexadecimal numbers because the pattern description requires 16 bits (and each hexadecimal number is 4 bits). Another example is a line of four foreground dots followed by four background dots. That PATTERN statement would be PATTERN &HFOFO.

By changing the COLOR statement to give the pens different colors, you can create a complete spectrum of patterned lines for your images. The pattern fill option works in a similar manner except that you define the pattern for a two-dimensional area. The "definition size" is still 16 bits wide, but is also *N* lines high. The *N* number must be a power of two — that is, 2, 4, 8, 16, 32, or 64.

The easiest way to define an area pattern is to put the hexadecimal statements into an *N*-dimensional array. The number of dimensions corresponds to the number of lines, and the number of elements in the array correspond to each line's pattern.

AMIGA TURTLE GRAPHICS

The following program creates a turtle graphics drawing language for your Amiga. To use this language, think of the cursor as a turtle having a pen in its mouth. Your commands can move the turtle to any position on the screen. If the pen is down, the turtle draws a line as it moves about the screen. If the pen is up, the turtle can still move but doesn't draw a line until you tell it to put the pen down.

The commands for this turtle graphics language are:

FD (*n*!) — moves turtle forward *n* pixels

BK (*n*!) — moves turtle backward *n* pixels

RT (*n*!) — rotates turtle *n* degrees to the right

LT (*n*!) — rotates turtle *n* degrees to the left

PU — picks pen up so it can move without drawing a line

PD — puts pen down so it draws as it moves

CS — erases all images on the screen and moves turtle to center of screen

Clean — erases images on the screen but leaves turtle at its position

SETPOS (x!,y!) — moves turtle to (x,y) coordinates

Towards (x!,y!) — points turtle to (x,y) coordinates

Home — repositions turtle at the center of the screen but does not erase images

Note that each number requires an exclamation point (!) after it.

To write programs, put them at the beginning of the code for the language. For example, a demonstration program called SpinBox is currently positioned at the appropriate place. After you've seen what SpinBox can do, erase it and put your own programs in the same place. You could also write your program and merge the graphics language to it using the Amiga BASIC MERGE command. MERGE adds a file to the end of another file so the language code would be in the right place.

Notice that each of the drawing commands requires a CALL statement. The language essentially defines a set of drawing routines with Amiga BASIC that you then call to get them to run. However, you can also mix other BASIC commands with the drawing commands. For example, the FOR...NEXT loop replicates a Logo function known as REPEAT, which repeats a command a specified number of times. Also, you could insert a COLOR statement to change the color of the pen and then draw multicolored images.

THE TURTLE GRAPHICS PROGRAM

```
    SCREEN 2, 320, 200, 2, 1
    WINDOW 2, "Turtle Graphics",,31,2
    CALL CS

REM This is the start of the SpinBox
REM demonstration program. Your program would
REM replace it here.

  GOSUB SpinBox

  CALL PU
  CALL SETPOS(-150!,0!)
  CALL PD
  CALL TOWARDS(-105!,45!)
  CALL FD(20!)
  FOR i! = 1! TO 1000!: NEXT i!
  CALL PE
  CALL BK(10!)
  CALL LT(45!)
  CALL PD
  CALL FD(50!)
  CALL CLEAN
  CALL HOME
  CALL RT(30!)
  CALL FD(40!)
```

```
END

SpinBox:
  FOR i = 1 TO 12
    GOSUB Box2
    CALL RT(30!)
  NEXT i
  RETURN

Box2:
  FOR j = 1 TO 4
    CALL FD(70!)
    CALL RT(90!)
  NEXT j
  RETURN

REM This is the end of the SpinBox program. You
REM would erase to this point and insert your
REM program. The remainder of the code is the
REM turtle graphics language. Each of the
REM subroutines is labeled for the command it
REM describes.

SUB CS STATIC
  SHARED XCOR, YCOR, HEADING, heading2, ps
  CLS
  XCOR = 0
  YCOR = 0
  HEADING = 0
  heading2 = 90
  ps = 1
END SUB

SUB FD(amt) STATIC
  SHARED XCOR, YCOR, heading2, ps
  lx = COS(heading2 * 3.14159/180)
  ly = SIN(heading2 * 3.14159/180)
  NX = XCOR + lx * amt
  ny = YCOR + ly * amt
  IF ps < 2 THEN LINE (XCOR+160,99-YCOR)-(NX+160,99-ny),ps
  XCOR = NX
  YCOR = ny
END SUB
```

```
SUB BK(amt) STATIC
  SHARED XCOR, YCOR, heading2, ps
  lx = COS(heading2 * 3.14159/180)
  ly = SIN(heading2 * 3.14159/180)
  NX = XCOR - lx * amt
  ny = YCOR - ly * amt
  IF ps ‹ 2 THEN LINE (XCOR+160,99-YCOR)-(NX+160,99-ny),ps
  XCOR = NX
  YCOR = ny
END SUB

SUB RT(amt) STATIC
  SHARED HEADING, heading2
  HEADING = (HEADING + amt) MOD 360
  heading2 = (450 - HEADING) MOD 360
END SUB

SUB LT(amt) STATIC
  SHARED HEADING, heading2
  HEADING = (HEADING - amt) MOD 360
  heading2 = (450 - HEADING) MOD 360
END SUB

SUB PD STATIC
  SHARED ps
  ps = 1
END SUB

SUB PU STATIC
  SHARED ps
  ps = 2
END SUB

SUB PE STATIC
  SHARED ps
  ps = 0
END SUB

SUB CLEAN STATIC
  CLS
END SUB

SUB SETPOS(x,y) STATIC
  SHARED XCOR, YCOR, PS
  IF ps ‹ 2 THEN LINE (XCOR + 160,99-YCOR)-(x + 160,99 -y),ps
  XCOR = x
  YCOR = y
END SUB

SUB TOWARDS(x,y) STATIC
  SHARED XCOR, YCOR, HEADING, heading2
  x1 = x - XCOR
  y1 = y - YCOR
  pi = 3.1415926535#
  pi2 = pi / 2
  IF ABS(x1) <= .000001 THEN
    a = pi2
    IF y1 < 0 THEN a = a + pi
    IF ABS(y1) < .000001 THEN a = 0
  ELSE
    a = ATN(y1/x1)
    IF x1 < 0 THEN a = a + pi
  END IF
  heading2 = (a * 180/pi) MOD 360
  HEADING = (450 - heading2) MOD 360
END SUB

SUB HOME STATIC
  SHARED XCOR, YCOR, HEADING, heading2, ps
  IF ps < 2 THEN LINE (XCOR+160,99-YCOR)- (160,99),ps
  HEADING = 0
  heading2 = 90
  XCOR = 0
  YCOR =0
END SUB
```

Why does this book waste so much of your time talking about Logo's turtle graphics when it's supposed to be a book about Amiga BASIC? The answer is that turtle graphics presents an alternate to Basic's way of looking at the world in Cartesian coordinates. Anything that can be done in the one can be done in the other, but often a solution to a problem will be much easier to solve in one of these systems than it will be in the other. This is an important point in computer programming. Being able to look at a problem in a variety of ways will make it easier for you to develop the best solution. Having a wide array of useful tools gives you that ability.

There is, of course, another far less philosophical reason for giving you a turtle graphics package. Many visually stunning programs have been written in Logo. Until a version of Logo becomes widely available for the

Amiga, knowing how (and having the tools) to convert a Logo program into an Amiga BASIC program will give you access to these. In that spirit, this book goes even further in showing how to convert from Logo to BASIC with the equivalency chart below. This is hardly complete (that would take a book all by itself) but if you know Logo and BASIC this should give you an excellant start at converting one to the other.

Logo	BASIC equivalent

IF ‹cond› [. . .]
(IF ‹cond› [. . .])
	IF ‹cond›
	THEN
	. . .
	ENDIF

IF ‹cond› [. . .] [. . .]
(IF ‹cond› [. . .] [. . .])
	IF ‹cond›
	THEN
	. . .
	ELSE
	. . .
	ENDIF
	‹cond› is a logical expression like (3 ‹ X) or (z$ = "TRUE").

REPEAT ‹count› [. . .]	FOR q = 1 TO ‹count› . . . NEXT q The index variable should be any one that wasn't otherwise used in the program. It is strictly a dummy variable. ‹count› is a numeric expression like 34 or (x * 5).
(LOCAL . . .)	This declares the listed variables to be local to the procedure. BASIC will do this automatically so any variables in the Logo procedure that aren't

	declared LOCAL (and which aren't formal parameters) should be put in a SHARED statement in the equivalent BASIC subroutine.
MAKE “var value	var = value
MAKE “var1 :var2	var1 = var2
:Q	This is a reference to the variable Q
TO name :var1 :var2. . . :varN	SUB name(var1,var2,. . .varN) STATIC
. . .	. . .
END	END SUB
TURTLE 60	CALL TURTLE(60!) This is how to convert calls to user-defined procedures.

Here is an example of a complete Logo program converted to Amiga BASIC:

```
TO TURTLE :SZ              SUB TURTLE(SZ!)  STATIC
(LOCAL “TSZ “P “Q)
                           ' The only global variables used are
                           ' xcor and ycor from the turtle
                           ' graphics routines. SZ! and TSZ! are
                           ' needed by the subfunctions.
                           SHARED xcor, ycor, SZ!, TSZ!
RT 18 PD                   CALL RT(18.0)
                           CALL PD
MAKE “TSZ :SZ / 3          TSZ! = SZ!/3
                           ' Looking at the Logo program, we see
                           ' that “P is used to store the current
                           ' position of the turtle. Since BASIC
                           ' doesn't have lists, it uses
                           ' two variables to store the current
                           ' X and Y coordinates. The same will
                           ' be true later on with “Q.
MAKE “P [ ]                PX! = -9999
                           PY! = -9999
```

```
REPEAT 5 [TURTLE1]                    FOR q = 1 TO 5
                                      NEXT q
REPEAT 6 [TURTLE2]                    FOR q = 1 TO 6
                                        CALL TURTLE2
                                      NEXT q
FD :TSZ LT 90 FD :SZ / 4 LT 18 PD     CALL FD(TSZ!)
                                      CALL LT(90.0)
                                      CALL FD(SZ!/4)
                                      CALL LT(18.0)
                                      CALL PD
REPEAT 5 [FD :SZ / 12 RT 72]          FOR q = 1 TO 5
                                        CALL FD(SZ!/12)
                                        CALL RT(72.0)
                                      NEXT q
PU RT 18 BK :SZ / 4 RT 90             CALL PU
                                      CALL RT(18.0)
                                      CALL BK(SZ!/4)
                                      CALL RT(90.0)
FD :TSZ LT 90 FD :SZ / 4 RT 18 PD     CALL FD(TSZ!)
                                      CALL LT(90.0)
                                      CALL FD(SZ!/4)
                                      CALL RT(18.0)
                                      CALL PD
REPEAT 5 [FD :SZ / 12 LT 72]          FOR q = 1 TO 5
                                        CALL FD(SZ!/12)
                                        CALL LT(72.0)
                                      NEXT q
PU LT 18 BK :SZ / 4 RT 90             CALL PU
                                      CALL LT(18.0)
                                      CALL BK(SZ!/4)
                                      CALL RT(90.0)
BK :TSZ * 2 LT 72 BK :SZ LT 18        CALL BK(TSZ!*2)
                                      CALL LT(72.0)
                                      CALL BK(SZ!)
                                      CALL LT (18.0)
END                                   END SUB

TO TURTLE1                            SUB TURTLE1 STATIC
                                      SHARED xcor, ycor, SZ!, TSZ!
FD :TSZ LT 108                        CALL FD(TSZ!)
                                      CALL LT(108.0)
REPEAT 5 [FD :TSZ RT 72]              FOR q = 1 TO 5
                                        CALL FD(TSZ!)
                                        CALL RT(72.0)
                                      NEXT q
```

```
RT 108 FD :TSZ * 2 RT 72                CALL RT(108.0)
                                        CALL FD(TSZ!*2)
                                        CALL RT(72.0)
END                                     END SUB

TO TURTLE2                              SUB TURTLE2 STATIC
                                        SHARED xcor, ycor, SZ!, TSZ!
FD :SZ / 2 RT 90 PD                     CALL FD(SZ!/2)
                                        CALL RT(90.0)
                                        CALL PD
FD :TSZ MAKE "Q POS                     CALL FD(TSZ!)
                                        ' xcor and ycor are global variables
                                        ' used by the turtle graphics routines
                                        ' to store the turtle's current X and
                                        ' Y positions.
                                        QX! = xcor
                                        QY! = ycor
IF :P = [ ]                             IF PX! = -9999
                                          THEN
  [MAKE "P POS]                             PX! = xcor
                                            PY! = ycor
                                          ELSE
  [SETPOS : P SETPOS :Q MAKE "P :Q]         CALL SETPOS (PX!,PY!)
                                            CALL SETPOS (QX!,QY!)
                                            PX! = QX!
                                            PY! = QY!
                                        ENDIF
PU BK :TSZ LT 90 FD :SZ / 2 RT 72       CALL PU
                                        CALL BK(TSZ!)
                                        CALL LT(90.0)
                                        CALL FD(SZ!/2)
                                        CALL RT(72.0)
END                                     END SUB
```

To try this out, put the TURTLE subprogram and the turtle graphics package into the following program and run it:

```
CLS
CALL INIT.TG
CALL TURTLE(60!)
END
‹‹TURTLE, TURTLE1, and TURTLE2 subprograms››
‹‹Turtle graphics package››
```

HINTS FOR SIMPLIFYING AMIGA GRAPHICS

1. Make a sketch of what you plan to draw. Knowing what you want to see on the screen makes writing the BASIC programs to do it much easier. Also, drawing the image first reveals where the programming might get complex. You can even trace complicated drawings onto clear plastic sheets and tape them to the screen to guide you.

2. If you plan to use coordinates, draw a template of the screen's position to help you figure out precise coordinates of each point. Even a hand-drawn set of horizontal and vertical lines helps to keep you oriented (Figure 5-4).

3. If you use the STEP option in your drawing statements, remember where the cursor is at all times. Shapes drawn with the STEP option are relative to the last cursor position.

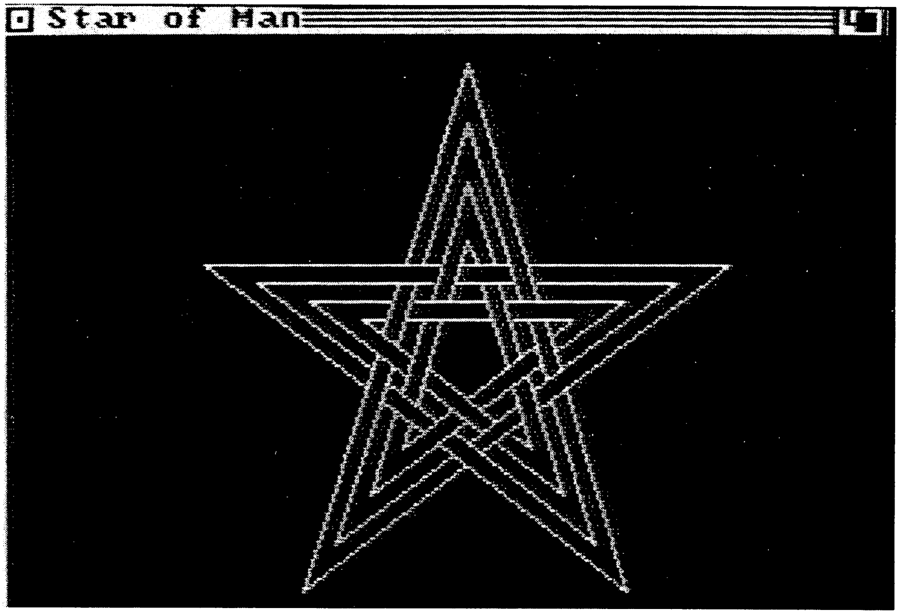


Figure 5-4

4. Draw a rough "blocked-out" image first, so you can visualize proper alignment and perspective for the scene. If you're new to artwork, find a reference book that explains the fundamentals of sketching.
5. To erase a portion of an image, draw over the portion with the pen set to the same color as the background.
6. Don't be discouraged if images get unexpectedly colored in shades you don't want. Different monitors display different shades of color. Plan to have your graphics garishly colored (or not at all) before you become proficient with the SCREEN and PALETTE statements.
7. Keep a table of PALETTE settings (red, green, and blue) for the color schemes that please you. Build a file of color tables that you can reuse with other programs.
8. Draw only a portion of an image and then run the program to see if it's right. If it is, go on to the next portion. If it isn't, correct it first and then go on. Although it may seem to take more time in the beginning, drawing incrementally like this makes writing the program much faster, especially when you have to find errors.
9. Modify previously written programs to draw more images. Changing a program you know to be correct can be far easier than creating a new one from scratch. Also, many images have portions that are useful as parts of other images. If you're serious about computer graphics, create a "shapes table" of programs that draw often-used forms (such as a star). Insert the programs creating the standard shapes into programs for drawing other images (for instance, a flag with a lot of stars on it).
10. If an image isn't quite right when it first appears on the screen, don't scrap the whole program. Chances are you'll be able to make a few minor editing corrections that will make your computer graphics turn out picture perfect.

CHAPTER 6

Animation

Animation has been around a long time. The first animated cartoon feature drawn for film was in 1906; and the first one to draw national attention appeared in 1909. It was a cartoon feature named *Gertie the Trained Dinosaur*, which its creator, Winsor McCay, used in a stage show. He projected the cartoon on a screen behind him and gave Gertie directions which, of course, she always followed. Audiences especially loved the last scene when Winsor threw Gertie an apple that she "caught" in her mouth.

Many other cartoonists followed McCay and created their own characters, but the next big breakthrough came in 1928 with Walt Disney's full-sound cartoon, *Steamboat Willie* (who was the ancestor to Mickey Mouse). By that time, animation had captured universal attention and appeal, which, if anything, is more pronounced today. Now, animation is ubiquitous, appearing in video games, TV ads, sports shows, and realistic movies such as the *Star Wars* trilogy. In fact, animation is so captivating that sometimes it's easy to forget how much work it takes to create good animation in the first place.

Ironically, the movie camera is part of the reason why many people underestimate the difficulty of creating animation. A movie camera automates the process that creates apparent movement of real objects. Each frame of the movie film is a still shot of a real scene, and when the frames are played in sequence at the proper speed, the result is apparent motion. Nothing is required except the ability to "point and shoot."

Movie animation is also shot with a camera and originally required an image drawn manually for each frame. With the advent of acetate overlays and other technical advances, however, animation artists have reduced the amount of actual drawing that goes into each animation series. The latest advance has been computers, which are used extensively in movie animation. From creating background scenes to mixing live action with animation, the computer brings a new dimension to animation. The Amiga's animation capabilities are like scaled-down versions of the movie computers, so you can produce moving images with a minimum of effort.

AMIGA ANIMATION OBJECTS

The Amiga supports two types of objects you can draw and move: one is known as a *BOB* and the other is a *sprite*. A sprite is a small image that can be moved around the screen by changing the coordinates of its upper left corner. You draw the image, give it a color, an initial position on the screen, and a velocity or acceleration to define its rate of movement.

The Amiga has eight "hardware" sprites, which means they are drawn and moved by special DMA circuitry in the animation chip. Hardware sprites can have three colors and be any height on the screen. They are limited to a maximum of 16 pixels wide and are only available in low resolution. Hardware sprites can also be transparent, which makes one object appear to pass under another one.

The other type of animation object is a BOB. BOB is an acronym for Blitter OBject. Like a sprite, you draw an image designated a BOB, give it a color, coordinates for a position, and velocity or acceleration to move it on the screen. Unlike a sprite, however, a BOB is controlled by a section of the animation chip called the Blitter. The term *Blitter* is short for "bit-blt" which itself is short for "bit-mapped block transfer," a formidable-sounding term that means the Amiga can move complete blocks of pixels on the screen from one position to another. Thus, a BOB is essentially a rectangular area of an image that you can "cut-and-paste" to another position on the screen.

A BOB acts like a sprite, but it has different capabilities. BOBs move slower but can be any size and, depending on the SCREEN definition, can be up to thirty-two colors at a time. Also, a scene can contain as many BOBs as the Amiga's memory can handle.

When creating animation with Amiga BASIC, you have the option of assigning the designation of BOB or sprite to each moving image. Then you use the OBJECT and COLLISION commands to describe how you want the action to occur on the screen. One note: sprites always appear in front of BOBs. Thus, if you want to draw a background scene, make it a BOB. You can designate which BOBs appear in front of others, but sprites always appear to be in the foreground.

HOW TO CREATE AN OBJECT FOR ANIMATION

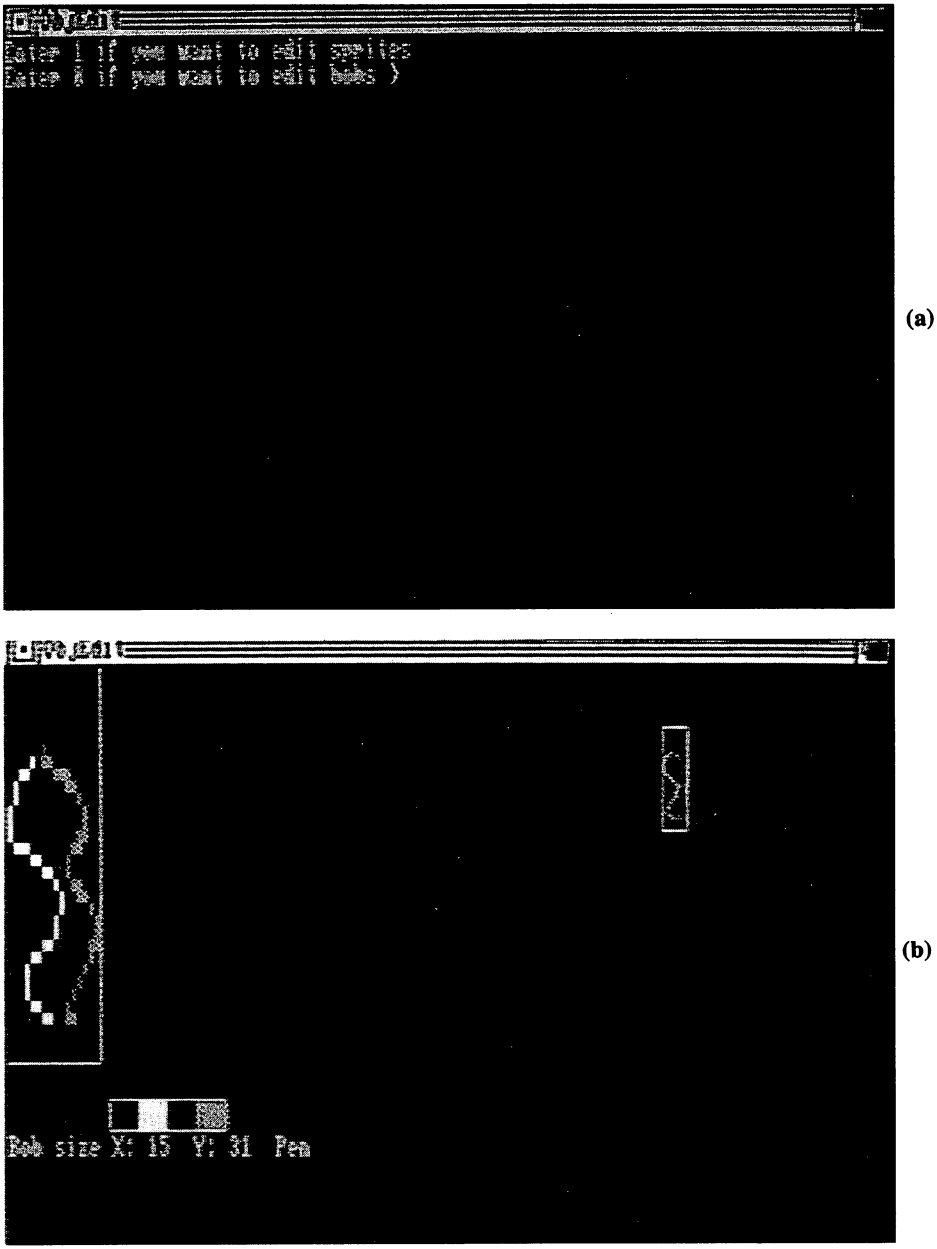
Amiga BASIC has a special routine, called OBJEDIT, for drawing and coloring animation objects. OBJEDIT is stored both on the Workbench disk and in a file on the Amiga BASIC disk. To load OBJEDIT from the Workbench, select the DEMOS icon and then select the OBJEDIT icon. Amiga BASIC automatically loads and starts the OBJEDIT routine. If you've already loaded Amiga BASIC, load the routine by typing:

```
LOAD "Basicdemos/OBJEDIT"
```

You can also select OPEN from the Project menu and then type Basicdemos/OBJEDIT. The routine's code appears on the window. Type RUN to start the OBJEDIT routine.

Immediately, you have a choice of making your drawing a sprite or a BOB. Enter either 1 or 0 and the OBJEDIT screen appears (Figure 6-1a and b). This is the screen or "canvas" for drawing images of objects that you want to animate. Press the right-hand mouse button to see the menu choices at the top of the screen. Select NEW from the Files menu to draw a new image; select OPEN to edit an existing image.

Select TOOLS and then select one of the six drawing options. You can mix the use of the tools. For instance, select OVAL for the basic shape of a



```
Enter 1 if you want to edit sprites
Enter 2 if you want to edit bobs >
```

```
Bob size X: 15  Y: 31  Pen
```

Figure 6-1

face, move the pointer to the place where you want the oval to appear, click the mouse button to draw it, then select PEN to sketch the face's features. Move the pointer to the place where the features belong, press the mouse button, and begin drawing.

You do the actual drawing by holding down the right mouse button. For instance, after selecting the PEN option, the mouse draws a line whenever the button is down. Similarly, selecting ERASE makes the mouse erase any portion of a drawn image when the button is down. Selecting the colors at the bottom of the canvas changes them on the drawn lines.

For drawing fine detail, select ENLARGE from the menu and select

Simple or Sequential Animation?

Simple animation moves an image about the screen without changing its orientation. The mouse pointer is an example of this concept. It moves but is always pointing in the same direction.

Sequential-animation images change their orientation as they move. An elementary example is a ball that appears to roll as it moves across the screen. More complex sequential animation is a person who seems to walk with arms swinging, legs alternately bending and straightening, and perhaps the head bobbing up and down.

To create sequential animation, you have two choices: draw separate images for each stage of the motion, or draw separate images for each part of the object that moves. Consider the person walking across the screen. In the first case you draw separate images for the entire person, with arms, legs, and head at different positions at each instant of motion. The illusion of motion is then created by displaying the images in their proper sequence and moving them one at a time to a new screen position. (If they didn't move, the person would "walk in place.")

In the second case you draw separate images for the right arm, left arm, right leg, left leg, head, and torso. Then, using the Amiga BASIC movement commands, you move each image separately, but staying in their proper positions to create the appearance of motion.

In either case, learning to draw the images with the right amount of movement between them comes with practice. Or you can refer to a book on animation for some hints. The most valuable and instructive books for learning the process are those that show original storyboards for old cartoons. Plus they're a lot of fun.

4x4 which enlarges the image by a factor of 4. Draw the details you want and then select 1x1 from the Enlarge menu. The image returns to its normal size with the fine details intact.

When the image is complete, select SAVE AS from the File menu to save a new image on a disk or SAVE to resave an existing image; give the image a filename to save it. Select QUIT from the File menu when you're done drawing and saving an image. If you loaded OBJEDIT from Amiga BASIC, the Output screen will reappear. If you loaded OBJEDIT from the Workbench disk, you must reload Amiga BASIC. Type NEW to use the animation commands.

The Amiga treats all of the drawing on one canvas as a single image. To have a scene show multiple images moving at the same time, draw separate images and save them as separate files. You can then refer to them individually and give each movement commands.

AMIGA BASIC STATEMENTS FOR ANIMATION

The special Amiga BASIC statements for animation fall into two groups: OBJECTS and COLLISIONS. The OBJECT statements move images; the COLLISION statements detect when two or more images collide and determine whether they are to pass through or bounce off each other.

You can also create animation with the Amiga BASIC GET and PUT statements. The GET command is similar to a BOB in that it transfers a rectangular area of the screen's graphics to a new location. The PUT command determines where the area is to be located on the screen.

The OBJECT Statements

After drawing an image on the animation canvas, it is, to Amiga BASIC, an object and can be manipulated with the OBJECT statements. The first step in writing an animation program is to give each image an identifying number. You do this with the OBJECT.SHAPE statement. But before you can assign it a number, you must load the image from its file to the program. The following program loads the image:

```
OPEN ‹filename› FOR INPUT AS 1
OBJECT.SHAPE n,INPUT$(LOF(1),1)
CLOSE 1
```

where the filename is the one you assigned the image after drawing it on the canvas and *n* is the identification number you want to assign to the object.

Repeat this program for each image file (if your animation is to have multiple objects), but change the filename and the number *n* each time. The images are then available to be animated. A hint: if the animation is to have many images moving together, keep a list of their identification numbers handy. OBJECT statements require that you identify separate objects.

OBJECT.SHAPE. This statement gives the object its identification number when you load its file. OBJECT.SHAPE also can duplicate existing objects. For example, if you've drawn a spaceship and assigned it an identification number of 2, but you want two of them in a scene, the statement OBJECT.SHAPE 3,2 copies object 2 as object 3. Object 3 is now another spaceship.

OBJECT.X and OBJECT.Y. These two statements position the top left corner of the image on the screen at an x- and y-coordinate. For obvious reasons, the statements almost always appear in pairs. For instance,

```
OBJECT.X 2,50
OBJECT.Y 2,100
```

puts object 2 at the position (50,100).

OBJECT.ON and OBJECT.OFF. Objects aren't visible on the screen until they're turned on. OBJECT.ON makes objects visible; OBJECT.OFF makes objects disappear. Thus,

```
OBJECT.OFF 4
OBJECT.ON 5
```

makes object 4 disappear and object 5 appear. You can turn on all the objects in a scene with a single OBJECT.ON not followed by any numbers.

One way to use the OBJECT.ON and OBJECT.OFF statements in sequential animation is to "stack" sequential images by giving them all the same set of starting coordinates, but turning on only the first image in the sequence. Then, using the OBJECT.VX and OBJECT.VY movement statements (described next), you move the entire stack to the next "frame" position, turn off the current object, and turn on the next one.

OBJECT.VX and OBJECT.VY. These two statements give an object a velocity in the x and y directions. Velocity is a constant speed and the x and y directions are horizontal and vertical. The velocity is measured in pixels per second. The statements

```
OBJECT.VX 2,30
OBJECT.VY 2,20
```

move object 2 in the x direction at 30 pixels per second and in the y direction at 20 pixels per second. With the screen set at low resolution (320 pixels in the x direction), it would take the object about 10 seconds to get from one side of the screen to the other. Similarly, with the screen set at noninterlaced (200 vertical lines), the object would also take about 10 seconds to traverse from the bottom to the top of the screen.

Positive numbers for the velocity move the object to the right or down; negative numbers move the object to the left or up.

OBJECT.AX and OBJECT.AY. Like the previous statements, these two also describe an object's motion, except the movement is an acceleration instead of a constant velocity. The acceleration is measured in terms of pixels per second per second (or seconds squared). The statements

```
OBJECT.AX 2,10
OBJECT.AY 2,20
```

move object 2 at an acceleration of 10 pixels per second squared in the x direction and 20 in the y direction. Note that the object will continue to speed up until you slow it down with an acceleration in the opposite directions (using negative numbers), or give it a constant velocity statement, or stop the object altogether with an OBJECT.STOP statement (which follows).

OBJECT.START, OBJECT.STOP, and OBJECT.CLOSE. Objects don't move until given an OBJECT.START statement, and they stay in motion until receiving an OBJECT.STOP or OBJECT.CLOSE statement, or until they collide with another object. You can start each object moving individually by typing OBJECT.START *n*, where *n* is the identifying number of the object. Or you can start all objects in a scene at the same time with a single OBJECT.START not followed by any numbers.

OBJECT.STOP *n* freezes an object (ID number *n*) at its current screen position. The object remains visible but doesn't move. If you're completely through with that object and no longer need it for the program, OBJECT.CLOSE *n* erases object *n* from the screen and clears it from memory. If you forget to specify an identification number with OBJECT.CLOSE, all defined objects are closed. Usually this erases everything from the screen.

OBJECT.PLANES. This statement is for changing the colors of the objects relative to the bit planes defined for their colors. It applies only to objects drawn as BOBs. The first parameter of the statement describes the sum of the bit values of the plane that you want to use for coloring the object, where bit values are 2 raised to the *nth* power and *n* is the plane number. For example, plane 0 has the bit value of 1, since 2 raised to the 0th power is 1, and plane 1 has a bit value of 2, since 2 raised to the 1st power is 2. To pick a new plane for determining pen colors, add the bit values for the bit planes you want. For instance, to display the object with the colors of bit planes 1 and 2, enter OBJECT.PLANES 1,6, where 1 is the object identification number and 6 is the sum (2 + 4) of the bit values for bit planes 1 (2) and 2 (4). The second parameter uses the same numbering scheme and adds another plane to the first parameter.

OBJECT.CLIP. This statement is for defining rectangles of the screen. No BOBs can then appear outside of the rectangle. The statement requires the coordinates of the top left and bottom right corners of the rectangle. For example, OBJECT.CLIP (10,10)-(100,100) creates a rectangle with the upper left corner at (10,10) and the bottom right at (100,100).

OBJECT.PRIORITY. When multiple BOBs are moving on the screen, two or more will probably intersect at some time. This OBJECT statement tells the Amiga which BOB is to pass in front of others. The statements

```
OBJECT.PRIORITY 3,4
OBJECT.PRIORITY 2,5
```

tell the Amiga that object 3's priority is 4 and object 2's priority is 5. Object 2 will always pass in front of object 3 because its priority is higher. Object 2 will also pass in front of all other objects with priorities from 0 to 4. This statement only applies to BOBs because sprites always pass in front of BOBs when they intersect.

OBJECT.HIT. When you have multiple objects in a scene, you may want some to collide when they intersect and others to pass under or over one another. You define the possible collisions with the OBJECT.HIT statement. To define collisions for an object, you give it three numbers. The first number is the object's identification number, the second number defines the object's MeMask, and the third number defines the object's HitMask. Think of the MeMask as the object's shield and the HitMask as its sword. If another object's sword penetrates the original object's shield, a collision occurs, but if the shield repels the sword, no collision occurs.

The two masks are actually 16 bits that specify the collision possibilities, and the way the Amiga determines "penetration and repulsion" of the swords and shields is by logically ANDing the bits of the HitMask of one object with the bits of the MeMask of another object. If the result of the logical AND is 0, no collision occurs, but if the result is not 0, a collision does occur. A logical AND states that the only time the combination of two binary numbers is 1 is when both are 1. Thus:

```
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0
1 AND 1 = 1
```

Putting this to work for a sample of two objects, suppose the HitMask of object 5 is the number 10 and the MeMask of object 6 is 4. The bit representation of 10 is 1010 and the bit representation of 4 is 0100. Logically ANDing the two bit patterns

```
1010
0100
----
0000
```

produces all 0's, so no collision occurs.

The statements for this situation are

```
OBJECT.HIT 5,n,10
OBJECT.HIT 6,4,n
```

where the *n* for object 5 is its MeMask number and the *n* for object 6 is its HitMask number. You would enter those *n* numbers depending on how you wanted those objects to collide with others.

Although this process may seem a little awkward, it provides you with a vast number of collision arrangements at little expense of the Amiga

HitMask and MeMask Simplified

Because HitMask and MeMask are confusing here's a simplified version that works if you have less than 16 objects on the screen.

Observe how the computer sees the two masks as binary numbers:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The MeMask for object 1 will be all zeros except at bit one, the MeMask for object 4 will be all zeros except at bit four, etc. Each mask will then have only one 1 bit in it. Now to select which objects an object will hit, simply put ones in each associated bit in the HitMask. For instance, to hit objects 14, 9, 7, 3, and 2 you would use 0100001010001100 which is &H428C. If you want an object to hit the border put a one in bit 0 of the HitMask.

As you gain more proficiency you will notice that it is possible to use MeMasks with multiple bits. This allows you to specify up to 64K different MeMasks. It also requires more complex HitMasks to determine what collides with what. Here is an example:

Given four objects A, B, C, and D which collide in the following ways:

A hits B or C
B hits A or D
C hits A or B
D hits A, B, or C

The MeMasks and HitMasks could be set up to use only three instead of four bits. This could be extended to the point where five sets of four-object relationships could be specified in the 15 bits available. The above relationships could use these masks (remember the zero bit is used for hitting the border):

	MeMask	HitMask
A	&H000A	&H0004
B	&H000C	&H0002
C	&H0004	&H000C
D	&H0002	&H0006

memory. After you get used to it, especially when multiple objects are in a scene, you'll see the advantages over trying to describe each possible collision. Initially it helps to draw the 0 and 1 bit patterns of the MeMasks and

HitMasks of all the objects in a scene in order to figure out the collisions and corresponding proper numbers for the OBJECT.HIT statement.

One exception to the logical ANDing is for collisions with the screen's border. If the least significant bit of an object's HitMask is set to 1, a collision always occurs with the border, regardless of the ANDing with another object's MeMask. Thus, for example, if an object's HitMask is 5, represented by the bit pattern 0101, the rightmost bit (which is the least significant bit) is 1, and the object will always collide with the screen's border. (For the sharp-eyed, it's obvious that odd numbers for the HitMask always have the least significant bit set to 1.)

Collisions

With the OBJECT.HIT and OBJECT.PRIORITY statements set, the Amiga can determine when collisions occur. Whenever two objects collide, two things happen: they stop and information about the collision is saved in stack memory. Information about subsequent collisions, up to sixteen, are also put into the stack. Any more collisions after the sixteenth are ignored. You can use the collision information in your programs to then cause something else to happen. For example, if the collision is between an asteroid and a spaceship, the program might jump to a scene of an explosion and a point counter that subtracts twenty-five from the previous total.

Various forms of the COLLISION statement reveal the collision information. The following three COLLISION statements are for detecting specific collisions.

COLLISION (0) tells the identification number of the object that has the information on the top of the stack. The information stays in the stack.

COLLISION (−1) lists the window number of the collision for the collision information on the top of the stack.

COLLISION (*n*), where *n* is an object identification number, lists a code that describes the type of collision that took place. A positive number identifies the object that collided with the object *n*; a 0 indicates no collision or that the collision is not the top one on the stack; negative numbers indicate collisions with screen borders — top border is −1, left −2, bottom −3, and collision with the right border −4.

You can also use the ON COLLISION statement to detect any collision and have the program branch when one occurs. Typically the form is

```
ON COLLISION GOSUB label
```

where *label* is the label of a subroutine in the program. ON COLLISION activates the collision detection and puts information in the stack, but for the program to be able to get that information and branch to a subroutine, the program must contain a COLLISION ON statement. After each collision the program will then detect collisions and branch as you've specified.

A COLLISION STOP statement deactivates the COLLISION ON statement but still puts information about subsequent collisions into the stack. Another COLLISION ON statement can restart the branching process. To completely stop detecting collisions, enter a COLLISION OFF statement in your program.

ANIMATION WITH THE GET AND PUT STATEMENTS

The Amiga BASIC GET and PUT statements can move objects around the screen in much the same manner as the OBJECT and COLLISION statements. The Amiga supports both types of graphics programming. GET and PUT are, of course, more familiar BASIC statements and have been around for a long time. The two work as a pair. GET defines a rectangle of an image by specifying the coordinates of its upper left and lower right corners, and then puts the rectangle into an array. The PUT statement transfers the image to its new position on the screen.

The following two programs show you how to use GET and PUT to play two video games. The first one is a rat's-eye view of a maze. You're the dismayed animal and must find your way through all the twists and turns. The images on the screen are as a rat would see them. Select the direction you want to go and the scene changes to show a new set of walls and doorways.

Type the program into a file, save it, then load it and type RUN. By the way, you can't learn the maze by playing the game. Each time it restarts, it draws a new maze. Give this game to any friends who are psychological research scientists.

The second program is a much easier game. It's a bird's-eye view of the maze and you move the rat through it by clicking on compass directions represented by the compass on the screen. (Try running the rat into a wall to see what he says.)

THE STRUCTURE OF THE MAZE PROGRAMS

Mazes are a staple of many video game programs. Dragons in castles, treasures guarded by fantasy creatures, or even working through facetious IRS crises, all have mazes as their underlying problem to solve. Sometimes, programmers draw the individual rooms of a maze which then stay the same throughout the game. In these two programs new mazes are drawn each time you start the games. Both programs use a single process to draw, and re-draw the rooms of the maze.

The main process for drawing the rooms is to treat them each as branches on a tree and to connect them according to a set of rules that determines if the rooms have doors opening to any other rooms. Because you are able to enter a room from one side, that leaves three other sides that can have doors, thus the tree structure is equivalent to a branch having at most three other branches emanating from it. Of course, in a maze rooms can have less than three doors open, therefore the next level of branches can number three, two, or one. (A tree with this structure is called a ternary tree.) Graphically this situation appears as shown in Figure 6-2.

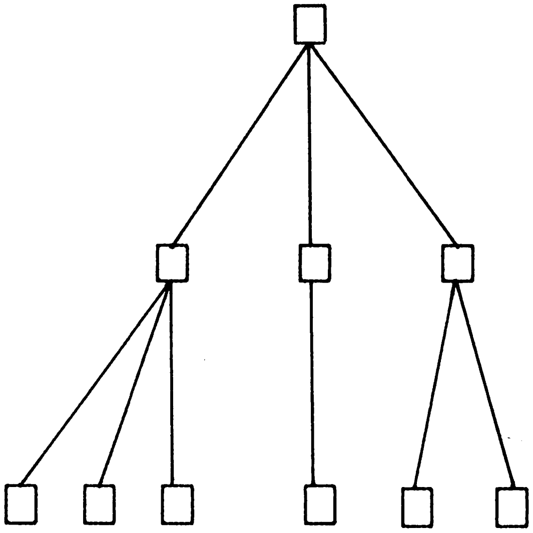


Figure 6-2

Each node in the tree is equivalent to a room. In the jargon of programming, each node has no more than four branches — 1 parent and three sons emanating from it — just as every room in the maze can have no more than four other rooms to which it attaches.

Before building the maze, the programs define an array for all the nodes of the tree, and initially define each node (room) in the array as being unattached. In the programs this is done by storing zero in every array element. Then the programs assign numbers describing how rooms are attached to other rooms, This is done with a numbering system where north, east, south, and west correspond to 1, 2, 4, and 8 (Figure 6-3).

Each time a door is added to a room, the door's value is added to the room's array value. Thus a room with a north, south, and west door would have the value 13 (1 + 4 + 8). The reason for choosing this numbering system is that it produces a unique bit code for each possible combination of doors. That is, when each of the four numbers is converted to binary, it has a single 1 bit distinct from the other three (0001, 0010, 0100, 1000) therefore, since each door number affects a seperate bit, every combination of doors is a unique value. Notice how similar this is to the simple method for specifying MeMasks and HitMasks. The more you program in BASIC, the more you will come to use these 'logical' operations or strings of bits. It is a VERY useful technique.

With this numbering system in place, the programs then pick any

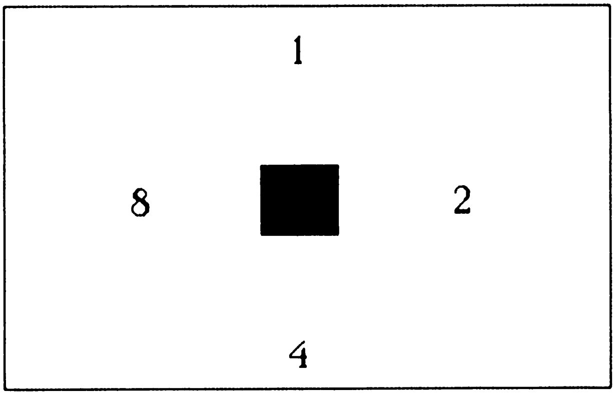


Figure 6-3

room at random and make it the starting room. This is the only room that must be handled in a special manner because no other rooms are attached to it and thus it must be assigned a positive number in the array. When any other room is added, it gets a positive value based on the direction in which it is attached to the already selected rooms (N,S, E, or W). But, because the first room isn't being attached to anything yet, it requires some arbitrary positive value. The number 9999 in the programs is the arbitrary number for the starting room. The number could be any positive value greater than 15. Why greater than 15? Because anything less describes one of the four door combinations — 15 is all four doors, 14 is west, east, and south doors, 13 is west, south and north doors, and so on.

The next step is to add a room to the tree and then look at that room's neighboring rooms. Any that have the value 0 receive the new value −1. These are called frontier cells and the −1 indicates they are still unattached but are adjacent to one or more rooms which are part of the tree. After the program selects the first room, only rooms adjacent to it may be chosen as the next room to be attached. In the array this means that only elements next to the selected element can be selected next. The program chooses one of them at random, connects it randomly to one of the attached rooms to which it is adjacent (which gives its array element a positive value), and then checks its four neighbors to see if they become frontier cells.

For example, to attach a room to its neighbor to the east, the program first puts a 2 in the room to indicate a door to the east, and then adds 8 to the value already in the neighbor room to indicate a door added to the west. Graphically, this would appear as shown in Figure 6-4.

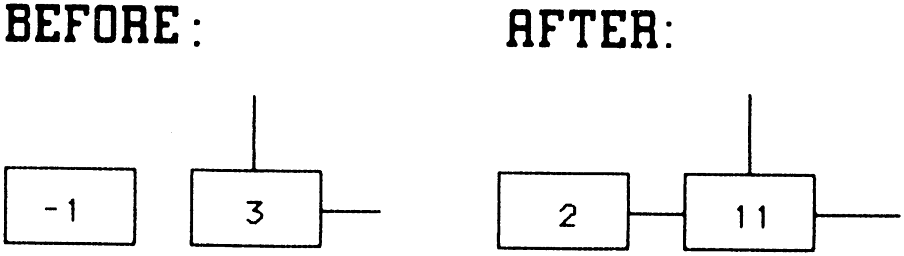


Figure 6-4

The programs have to find all frontier cells and add rooms to them. The programs pick any cell (element in the array) and determine if it's a frontier (has a value of −1). If it is, the rooms and doors are built around it. If it isn't, the programs move vertically through the array's columns until finding the next frontier cell. By keeping count of the number of elements and continuing through the array only as long as the count is non-zero, the programs find all the frontier cells, add rooms and doors to them, and then stop. At this point the maze is essentially complete. All that remains is to subtract the first room's special value (making it just an ordinary room like any other), and to choose any room in the first column of the array for an entrance and any room in the last column for an exit.

You can choose any room in the appropriate columns for the entrance and exit because going through the maze is exactly equivalent to traversing the tree. As there is a unique path from any tree node to any other tree node, so there is only one path from any room chosen as the entrance to any room chosen as the exit.

So what do the programs have after this is all over? An array of rooms with values indicating which walls in each room have doors. This allows the programs to check if a door exists in the direction the player wants to move. If so, the player moves in that direction and the next room's doors are available. If no door exists in the direction, the program can honk, beep, and complain.

The easiest way to test for the presence of a door is to combine with the logical AND function, the value of the room against the value of a door going in that direction. If the room has a door in that direction, the value will be returned. If it does not, a zero will be returned. Thus, for example, (13 AND 4) returns 4 but (13 AND 2) returns 0.

The fact that both programs use this same structure for both the bird's-eye and rat's-eye view mazes illustrates how graphics programs rely on specific methods for drawing certain elements. This particular method is a technique used in a number of commercial video maze games and, with some slight modifications could map a maze from a doughnut shaped image to a rectangle. Commercial game programmers take great delight in making different levels of a maze increasingly complex both in programming and in solving. The Amiga's high resolution, multilevel graphics system is ideal for maze games.

THE RAT'S-EYE VIEW MAZE PROGRAM

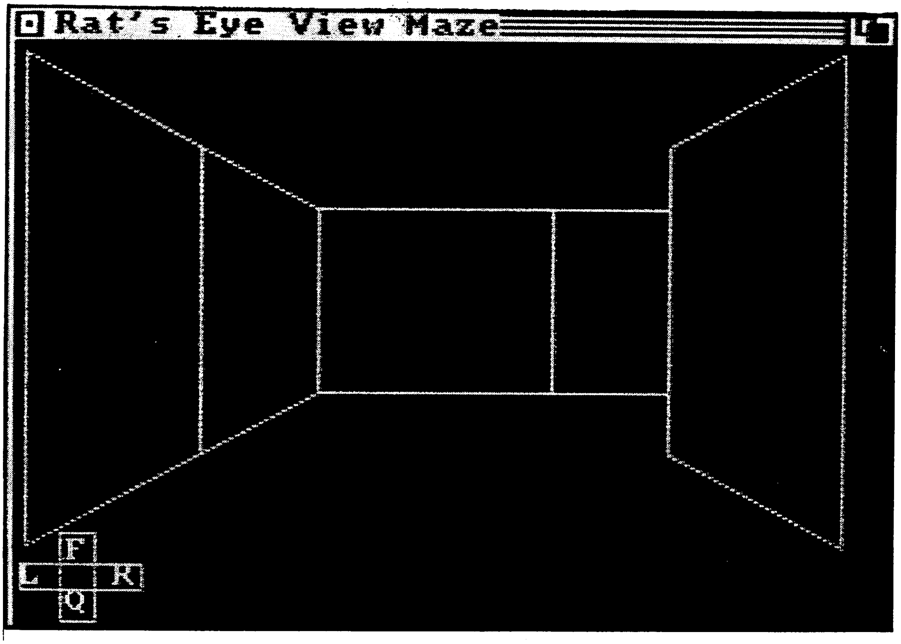


Figure 6-5

```
'
' Rat's-eye View Maze
'

maze:
  SCREEN 2, 320, 200, 2, 1
  WINDOW 2, "Rat's-eye View Maze",,31,2

  DEFINT a-z
  DEF FNR(x) = INT(RND * x)
  RANDOMIZE TIMER

  xMax = 8
  yMax = 8

  true = (1 = 1)
  false = NOT true
  GameWanted = true

DIM maze(xMax,yMax), offset1(15), offset2(15)
DIM Wall(15), xpos(8), ypos(8)

'
' Initialize arrays
'
FOR i% = 1 TO 15
  READ offset1(i%), offset2(i%)
NEXT i%

DATA -1, 2, 0, 2, 1, 2, 0, 2, 0, 2
DATA -1, 1, 0, 1, 1, 1, 0, 1, 0, 1
DATA -1, 0, 0, 0, 1, 0, 0, 0, 0, 0

FOR i% = 1 TO 8
  READ xpos(i%), ypos(i%)
NEXT i%

DATA 2, 2, 62, 32, 102, 52, 122, 62
DATA 162, 102, 182, 112, 222, 132, 282, 162

WHILE (GameWanted)
  GOSUB GenerateMaze
  GOSUB TraverseMaze
  GOSUB AskForAnotherGame
WEND
WINDOW OUTPUT 1
WINDOW CLOSE 2
STOP

END

'
' Maze Generation Subroutine
'   This algorithm converts the array into a randomly branching
'   binary tree. Going through the maze then becomes a matter of
'   simply traversing the tree.
'
GenerateMaze:
  FrontierNum = 0
  PRINT "Building the maze"
  ‘
  ‘ Clear all maze cells
  ‘
FOR x = 0 TO xMax
  FOR y = 0 TO yMax
    maze(x,y) = 0
  NEXT y
NEXT x

‘
‘ Choose an initial cell in the binary tree
‘
xInit = FNR(xMax+1)
yInit = FNR(yMax+1)
maze(xInit,yInit) = 9999
CALL Frontiers(xInit,yInit)

‘
‘ Add cells to the binary tree until all maze cells are joined
‘
WHILE (FrontierNum › 0)

  FindFrontier:
  xFirst = FNR(xMax+1)
  yFirst = FNR(yMax+1)
  x = xFirst
  y = yFirst

  WHILE (maze(x,y) ‹› −1)
    IF x = xMax THEN x = 0 ELSE x = x + 1
    IF x = xFirst THEN
      IF y = yMax THEN y = 0 ELSE y = y + 1
    END IF
  WEND

ConnectCell:
  FrontierNum = FrontierNum − 1
  side = FNR(4) + 1
  q = −1

  WHILE (q ‹ 1)
    IF side = 4 THEN side = 1 ELSE side = side + 1
    ON side GOTO CC1, CC2, CC3, CC4
  CC1:
    IF x › 0 THEN q = maze(x-1,y)
    GOTO CC5
  CC2:
    IF x ‹ xMax THEN q = maze(x+1,y)
    GOTO CC5
  CC3:
    IF y › 0 THEN q = maze(x,y-1)
    GOTO CC5
  CC4:
    IF y ‹ yMax THEN q = maze(x,y+1)
    GOTO CC5
  CC5:
  WEND

ON side GOTO CC11, CC12, CC13, CC14
CC11:
    maze(x-1,y) = maze(x-1,y) + 2
    maze(x,y) = 8
    GOTO CC15
  CC12:
    maze(x+1,y) = maze(x+1,y) + 8
    maze(x,y) = 2
    GOTO CC15
  CC13:
    maze(x,y-1) = maze(x,y-1) + 4
    maze(x,y) = 1
    GOTO CC15
  CC14:
    maze(x,y+1) = maze(x,y+1) + 1
    maze(x,y) = 4
    GOTO CC15
  CC15:
  CALL Frontiers(x,y)

WEND

'
' Now readjust the initial cell in the binary tree
'
maze(xInit,yInit) = maze(xInit,yInit) - 9999

'
' Choose entrance and exit cells
'
xIn = 0
yIn = FNR(yMax)
```

```
xOut = xMax
yOut = FNR(yMax)
maze(xOut,yOut) = maze(xOut,yOut) + 2

‘
‘ End of GenerateMaze subroutine
‘
RETURN

‘
‘ This subprogram checks to see which adjacent cells become
‘ frontier cells. It also adjusts the frontier cell counter.
‘
SUB Frontiers(x,y) STATIC
  SHARED xMax, yMax, FrontierNum, maze()
  f = FrontierNum
  IF x › 0    THEN IF maze(x-1,y) = 0  THEN maze(x-1,y) = -1:f = f + 1
  IF x ‹ xMax THEN IF maze(x+1,y) = 0  THEN maze(x+1,y) = -1:f = f + 1
  IF y › 0    THEN IF maze(x,y-1) = 0  THEN maze(x,y-1) = -1:f = f + 1
  IF y ‹ yMax THEN IF maze(x,y+1) = 0  THEN maze(x,y+1) = -1:f = f + 1
FrontierNum = f
END SUB

’
’ Player moves the mouse through the maze
’
TraverseMaze:
  x = xIn
  y = yIn
  GameOver = false
  direction = 2 ’ east
  CLS

’
’ Draw the direction-compass
’
LOCATE 21,3: PRINT “F”;
LOCATE 22,1: PRINT “L R”;
LOCATE 23,3: PRINT “Q”;
LINE (14,158)-(26,186),1,b
LINE (0,168)-(42,176),1,b

kolor = 1
GOSUB Map
```

```
WHILE (NOT GameOver)
  GOSUB GetMove

  IF move = 1 THEN
    kolor = 0
    GOSUB Map
    IF direction = 0 THEN
    IF (maze(x,y) AND 1) = 1 THEN y = y −1 ELSE GOSUB Complain
  END IF
  IF direction = 1 THEN
    IF (maze(x,y) AND 4) = 4 THEN y = y + 1 ELSE GOSUB Complain
  END IF
  IF direction = 2 THEN
    IF (maze(x,y) AND 2) = 2 THEN x = x + 1 ELSE GOSUB Complain
  END IF
  IF direction = 3 THEN
    IF (maze(x,y) AND 8) = 8 THEN x = x −1 ELSE GOSUB Complain
  END IF
  kolor = 1
  GOSUB Map
END IF

IF move = 2 THEN GameOver = true

IF move = 3 THEN
    kolor = 0
    GOSUB Map
    direction = VAL(MID$(“2310”,direction+1,1))
    kolor = 1
    GOSUB Map
  END IF

  IF move = 4 THEN
    kolor = 0
    GOSUB Map
    direction = VAL(MID$(“3201”,direction+1,1))
    kolor = 1
    GOSUB Map
END IF

  IF (x = xMax + 1) THEN
    GameOver = true
    SAY TRANSLATE$(“Exit, stage right!”)
  END IF
```

```
WEND
RETURN

'
' GetMove waits for the user to click on one of the directions
' in the compass
'
GetMove:

  dx = -1
  dy = -1
  WHILE (MOUSE(0) ‹ 0)
    dx = MOUSE(1)
    dy = MOUSE(2)
  WEND

  IF dx = -1 THEN GetMove

  IF (dx›14) AND (dx‹26) AND (dy›158) AND (dy‹170) THEN
    move = 1
    RETURN
  END IF

  IF (dx›14) AND (dx‹26) AND (dy›174) AND (dy‹186) THEN
    move = 2
    RETURN
  END IF

  IF (dx›-1) AND (dx‹11) AND (dy›166) AND (dy‹177) THEN
    move = 4
    RETURN
  END IF

  IF (dx›29) AND (dx‹41) AND (dy›166) AND (dy‹177) THEN
    move = 3
    RETURN
  END IF

  GOTO GetMove

'
' See if the player wants to try again
'
AskForAnotherGame:
  CLS
  PRINT
  PRINT “ Would you like to play again ”;
  INPUT a$
  a$ = LEFT$(a$,1)
  GameWanted = ((a$ = “Y”) OR (a$ = “y”))
  RETURN

Complain:
  SAY TRANSLATE$(“Ouch!”)
  RETURN

'
' This subroutine determines the walls that are visible.
' It does this by finding which walls are present and then
' following a logic chain telling which walls block which other
' walls from sight.
'
Map:
  GOSUB Walls

  IF Wall(12) THEN
    CALL WallDraw(12,kolor)
  ELSE
    IF (x = xOut) AND (y = yOut) AND (direction = 2) THEN
      LOCATE 12,15
      PRINT “EXIT!”;
  END IF
  IF Wall(7) THEN
    CALL WallDraw(7,kolor)
  ELSE
    IF Wall(2) THEN CALL WallDraw(2,kolor)
  END IF
END IF

IF Wall(14) THEN
  CALL WallDraw(14,kolor)
ELSE
  IF Wall(11) THEN CALL WallDraw(11,kolor)
END IF

IF Wall(15) THEN
  CALL WallDraw(15,kolor)
ELSE
  IF Wall(13) THEN CALL WallDraw(13,kolor)
END IF
```

```
IF NOT Wall(12) THEN
  IF Wall(9) THEN
    CALL WallDraw(9,kolor)
  ELSE
    IF Wall(6) THEN CALL WallDraw(6,kolor)
  END IF
  IF Wall(10) THEN
    CALL WallDraw(10,kolor)
  ELSE
    IF Wall(8) THEN CALL WallDraw(8,kolor)
  END IF
END IF

IF NOT (Wall(12) OR Wall(7)) THEN
  IF Wall(4) THEN
    CALL WallDraw(4,kolor)
  ELSE
    IF Wall(1) THEN CALL WallDraw(1,kolor)
  END IF
  IF Wall(5) THEN
    CALL WallDraw(5,kolor)
  ELSE
    IF Wall(3) THEN CALL WallDraw(3,kolor)
  END IF
 END IF
RETURN

Walls:
  FOR n = 1 TO 15
    ON direction + 1 GOTO W0, W1, W2, W3
   W0:
    rx = x + offset1(n)
    ry = y - offset2(n)
    mask = 1
    IF (n=4) OR (n=9) OR (n=14) THEN mask = 8
    IF (n=5) OR (n=10) OR (n=15) THEN mask = 2
    GOTO W4
   W1:
    rx = x - offset1(n)
    ry = y + offset2(n)
    mask = 4
    IF (n=4) OR (n=9) OR (n=14) THEN mask = 2
    IF (n=5) OR (n=10) OR (n=15) THEN mask = 8
    GOTO W4
   W2:
    rx = x + offset2(n)
    ry = y + offset1(n)
    mask = 2
    IF (n=4) OR (n=9) OR (n=14) THEN mask = 1
    IF (n=5) OR (n=10) OR (n=15) THEN mask = 4
    GOTO W4
   W3:
    rx = x - offset2(n)
    ry = y - offset1(n)
    mask = 8
    IF (n=4) OR (n=9) OR (n=14) THEN mask = 4
    IF (n=5) OR (n=10) OR (n=15) THEN mask = 1
    GOTO W4
   W4:
    IF (rx‹0) OR (rx›xMax) OR (ry‹0) OR (ry›yMax) THEN
      Wall(n) = true
    ELSE
      Wall(n) = ((maze(rx,ry) AND mask) = 0)
    END IF
  NEXT n
RETURN

'
' This routine does the actual wall drawing
'
SUB WallDraw(n,c) STATIC
  SHARED xpos(), ypos()

  ON n GOTO
wd1,wd2,wd3,wd4,wd5,wd6,wd7,wd8,wd9,wd10,wd11,wd12,wd13,wd14,wd15
  wd1:
    LINE (xpos(3),ypos(5))-(xpos(3),ypos(4)),c
    LINE (xpos(3),ypos(4))-(xpos(4),ypos(4)),c
    LINE (xpos(4),ypos(4))-(xpos(4),ypos(5)),c
    LINE (xpos(4),ypos(5))-(xpos(3),ypos(5)),c
    EXIT SUB
  wd2:
    LINE (xpos(4),ypos(5))-(xpos(4),ypos(4)),c
    LINE (xpos(4),ypos(4))-(xpos(5),ypos(4)),c
    LINE (xpos(5),ypos(4))-(xpos(5),ypos(5)),c
    LINE (xpos(5),ypos(5))-(xpos(4),ypos(5)),c
    EXIT SUB
  wd3:
    LINE (xpos(5),ypos(5))-(xpos(5),ypos(4)),c
   LINE (xpos(5),ypos(4))-(xpos(6),ypos(4)),c
   LINE (xpos(6),ypos(4))-(xpos(6),ypos(5)),c
   LINE (xpos(6),ypos(5))-(xpos(5),ypos(5)),c
   EXIT SUB
wd4:
   LINE (xpos(3),ypos(6))-(xpos(3),ypos(3)),c
   LINE (xpos(3),ypos(3))-(xpos(4),ypos(4)),c
   LINE (xpos(4),ypos(4))-(xpos(4),ypos(5)),c
   LINE (xpos(4),ypos(5))-(xpos(3),ypos(6)),c
   EXIT SUB
wd5:
   LINE (xpos(5),ypos(5))-(xpos(5),ypos(4)),c
   LINE (xpos(5),ypos(4))-(xpos(6),ypos(3)),c
   LINE (xpos(6),ypos(3))-(xpos(6),ypos(6)),c
   LINE (xpos(6),ypos(6))-(xpos(5),ypos(5)),c
   EXIT SUB
wd6:
   LINE (xpos(2),ypos(6))-(xpos(2),ypos(3)),c
   LINE (xpos(2),ypos(3))-(xpos(3),ypos(3)),c
   LINE (xpos(3),ypos(3))-(xpos(3),ypos(6)),c
   LINE (xpos(3),ypos(6))-(xpos(2),ypos(6)),c
   EXIT SUB
wd7:
   LINE (xpos(3),ypos(6))-(xpos(3),ypos(3)),c
   LINE (xpos(3),ypos(3))-(xpos(6),ypos(3)),c
   LINE (xpos(6),ypos(3))-(xpos(6),ypos(6)),c
   LINE (xpos(6),ypos(6))-(xpos(3),ypos(6)),c
   EXIT SUB
wd8:
   LINE (xpos(6),ypos(6))-(xpos(6),ypos(3)),c
   LINE (xpos(6),ypos(3))-(xpos(7),ypos(3)),c
   LINE (xpos(7),ypos(3))-(xpos(7),ypos(6)),c
   LINE (xpos(7),ypos(6))-(xpos(6),ypos(6)),c
   EXIT SUB
wd9:
   LINE (xpos(2),ypos(7))-(xpos(2),ypos(2)),c
   LINE (xpos(2),ypos(2))-(xpos(3),ypos(3)),c
   LINE (xpos(3),ypos(3))-(xpos(3),ypos(6)),c
   LINE (xpos(3),ypos(6))-(xpos(2),ypos(7)),c
   EXIT SUB
wd10:
   LINE (xpos(6),ypos(6))-(xpos(6),ypos(3)),c
   LINE (xpos(6),ypos(3))-(xpos(7),ypos(2)),c
   LINE (xpos(7),ypos(2))-(xpos(7),ypos(7)),c
   LINE (xpos(7),ypos(7))-(xpos(6),ypos(6)),c
```

```
    EXIT SUB
  wd11:
    LINE (xpos(1),ypos(7))-(xpos(1),ypos(2)),c
    LINE (xpos(1),ypos(2))-(xpos(2),ypos(2)),c
    LINE (xpos(2),ypos(2))-(xpos(2),ypos(7)),c
    LINE (xpos(2),ypos(7))-(xpos(1),ypos(7)),c
    EXIT SUB
  wd12:
    LINE (xpos(2),ypos(7))-(xpos(2),ypos(2)),c
    LINE (xpos(2),ypos(2))-(xpos(7),ypos(2)),c
    LINE (xpos(7),ypos(2))-(xpos(7),ypos(7)),c
    LINE (xpos(7),ypos(7))-(xpos(2),ypos(7)),c
    EXIT SUB
  wd13:
    LINE (xpos(7),ypos(7))-(xpos(7),ypos(2)),c
    LINE (xpos(7),ypos(2))-(xpos(8),ypos(2)),c
    LINE (xpos(8),ypos(2))-(xpos(8),ypos(7)),c
    LINE (xpos(8),ypos(7))-(xpos(7),ypos(7)),c
    EXIT SUB
  wd14:
    LINE (xpos(1),ypos(8))-(xpos(1),ypos(1)),c
    LINE (xpos(1),ypos(1))-(xpos(2),ypos(2)),c
    LINE (xpos(2),ypos(2))-(xpos(2),ypos(7)),c
    LINE (xpos(2),ypos(7))-(xpos(1),ypos(8)),c
    EXIT SUB
  wd15:
    LINE (xpos(7),ypos(7))-(xpos(7),ypos(2)),c
    LINE (xpos(7),ypos(2))-(xpos(8),ypos(1)),c
    LINE (xpos(8),ypos(1))-(xpos(8),ypos(8)),c
    LINE (xpos(8),ypos(8))-(xpos(7),ypos(7)),c
    EXIT SUB
END SUB
```

THE BIRD'S-EYE VIEW MAZE PROGRAM

To see what the maze looks like, see Figure 6-6.

```
'
' Bird's-eye View Maze
'
maze:

  SCREEN 2, 320, 200, 2, 1
  WINDOW 2, "Rats!",,31,2
```

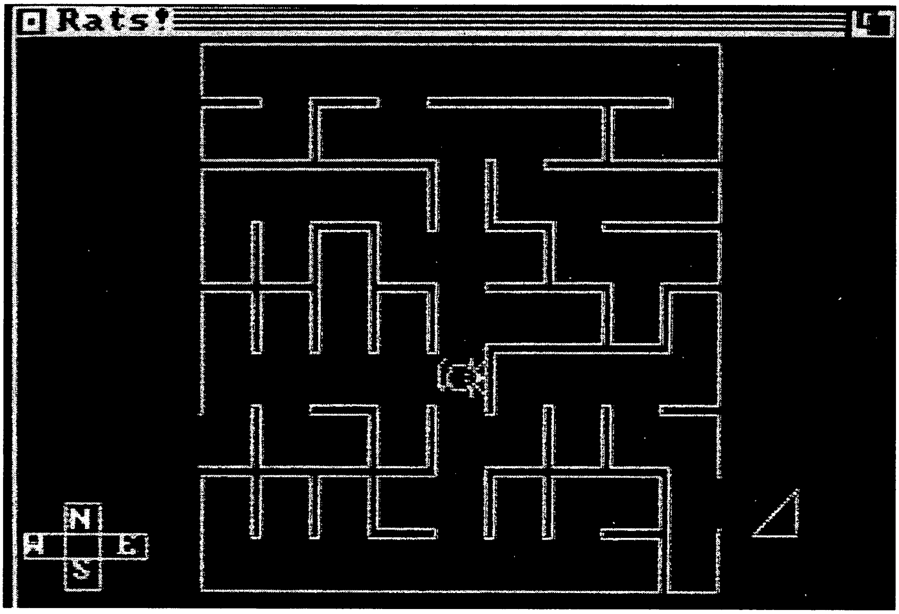


Figure 6-6

```
   DEFINT a-z
   DEF FNR(x) = INT(RND * x)
   RANDOMIZE TIMER

   xMax = 8
   yMax = 8

   true = (1 = 1)
   false = NOT true
   GameWanted = true
   DIM maze(xMax,yMax)

GOSUB InitMice

WHILE (GameWanted)
   GOSUB GenerateMaze
   GOSUB PrintMaze
```

```
  GOSUB TraverseMaze
  GOSUB AskForAnotherGame
WEND

WINDOW OUTPUT 1
WINDOW CLOSE 2
STOP

END

'
' Maze Generation Subroutine
'   This algorithm converts the array into a randomly branching
'   binary tree. Going through the maze then becomes a matter of
'   simply traversing the tree.
'


GenerateMaze:
  FrontierNum = 0
  PRINT "Building the maze"

  ‘
  ‘ Clear all maze cells
  ‘
  FOR x = 0 TO xMax
    FOR y = 0 TO yMax
      maze(x,y) = 0
    NEXT y
  NEXT x

  ‘
  ‘ Choose an initial cell in the binary tree
  ‘
  xInit = FNR(xMax+1)
  yInit = FNR(yMax+1)
  maze(xInit,yInit) = 9999
  CALL Frontiers(xInit,yInit)

  ‘
  ‘ Add cells to the binary tree until all maze cells are joined
  ‘
  WHILE (FrontierNum › 0)

   FindFrontier:
    xFirst = FNR(xMax+1)
    yFirst = FNR(yMax+1)
    x = xFirst
    y = yFirst

    WHILE (maze(x,y) <> -1)
      IF x = xMax THEN x = 0 ELSE x = x + 1
      IF x = xFirst THEN
        IF y = yMax THEN y = 0 ELSE y = y + 1
    END IF
  WEND

ConnectCell:
  FrontierNum = FrontierNum - 1
  side = FNR(4) + 1
  q = -1

  WHILE (q < 1)
    IF side = 4 THEN side = 1 ELSE side = side + 1
    ON side GOTO CC1, CC2, CC3, CC4
    CC1:
        IF x > 0 THEN q = maze(x-1,y)
        GOTO CC5
    CC2:
        IF x < xMax THEN q = maze(x+1,y)
        GOTO CC5
    CC3:
        IF y > 0 THEN q = maze(x,y-1)
        GOTO CC5
    CC4:
        IF y < yMax THEN q = maze(x,y+1)
        GOTO CC5
    CC5:
    WEND

    ON side GOTO CC11, CC12, CC13, CC14
    CC11:
        maze(x-1,y) = maze(x-1,y) + 2
        maze(x,y) = 8
        GOTO CC15
    CC12:
        maze(x+1,y) = maze(x+1,y) + 8
        maze(x,y) = 2
        GOTO CC15
    CC13:
        maze(x,y-1) = maze(x,y-1) + 4
        maze(x,y) = 1
        GOTO CC15
    CC14:
        maze(x,y+1) = maze(x,y+1) + 1
        maze(x,y) = 4
        GOTO CC15
    CC15:
    CALL Frontiers(x,y)

WEND

'
' Now readjust the initial cell in the binary tree
'
maze(xInit,yInit) = maze(xInit,yInit) - 9999

'
' Choose entrance and exit cells
'
xIn = 0
yIn = FNR(yMax)
maze(xIn,yIn) = maze(xIn,yIn) + 8
xOut = xMax
yOut = FNR(yMax)
maze(xOut,yOut) = maze(xOut,yOut) + 2

'
' End of GenerateMaze subroutine
'
RETURN

'
' Maze Display Routine
'   Draw the maze and the wedge of cheese which is the goal.
'   Also draw the direction-command compass.
'
PrintMaze:
  CLS
  Csize = 20
  TC = 1

  FOR y = 0 TO yMax
  yb = y * Csize
  ye = yb + Csize - 1
  FOR x = 0 TO xMax
```

```
    xb = x * Csize + 60
    xe = xb + Csize - 1

  IF (maze(x,y) AND 8) = 8 THEN
    LINE (xb,yb+TC)-(xb+TC,yb+TC),1
    LINE (xb,ye-TC)-(xb+TC,ye-TC),1
  ELSE
    LINE (xb+TC,yb+TC)-(xb+TC,ye-TC),1
  END IF

  IF (maze(x,y) AND 4) = 4 THEN
    LINE (xb+TC,ye-TC)-(xb+TC,ye),1
    LINE (xe-TC,ye-TC)-(xe-TC,ye),1
  ELSE
    LINE (xb+TC,ye-TC)-(xe-TC,ye-TC),1
  END IF

  IF (maze(x,y) AND 2) = 2 THEN
    LINE (xe-TC,ye-TC)-(xe,ye-TC),1
    LINE (xe-TC,yb+TC)-(xe,yb+TC),1
  ELSE
    LINE (xe-TC,yb+TC)-(xe-TC,ye-TC),1
  END IF

  IF (maze(x,y) AND 1) = 1 THEN
    LINE (xb+TC,yb+TC)-(xb+TC,yb),1
    LINE (xe-TC,yb+TC)-(xe-TC,yb),1
  ELSE
    LINE (xb+TC,yb+TC)-(xe-TC,yb+TC),1
  END IF
 NEXT x
NEXT y

'
' Draw the wedge of cheese
'
x = (xOut+1)*Csize + 70
y = (yOut+1)*Csize
LINE (x,y)-(x+15,y-15),1
LINE (x+15,y-15)-(x+15,y),1
LINE (x+15,y)-(x,y),1

'
' Draw the direction-compass
'
```

```
LOCATE 20,3: PRINT "N";
LOCATE 21,1: PRINT "W     E";
LOCATE 22,3: PRINT "S";
LINE (14,150)-(26,178),1,b
LINE (0,160)-(42,168),1,b

RETURN

'
' This subprogram checks to see which adjacent cells become
' frontier cells. It also adjusts the frontier cell counter.
'

SUB Frontiers(x,y) STATIC
  SHARED xMax, yMax, FrontierNum, maze()
  f = FrontierNum
  IF x > 0    THEN IF maze(x-1,y) = 0  THEN maze(x-1,y) = -1:f = f + 1
  IF x < xMax THEN IF maze(x+1,y) = 0  THEN maze(x+1,y) = -1:f = f + 1
  IF y > 0    THEN IF maze(x,y-1) = 0  THEN maze(x,y-1) = -1:f = f + 1
  IF y < yMax THEN IF maze(x,y+1) = 0  THEN maze(x,y+1) = -1:f = f + 1
  FrontierNum = f
END SUB

'
' Draw and store mouse in four directions
'
InitMice:
  DIM   mdata(15,15),   mouseN(100),   mouseS(100),   mouseE(100),
mouseW(100)

  FOR y = 0 TO 15
    FOR x = 0 TO 15
      READ mdata(x,y)
    NEXT x
  NEXT y

  DATA 0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0
  DATA 0,0,0,1,0,0,1,1,1,0,0,1,0,0,0,0
  DATA 0,0,0,0,1,0,1,1,1,0,1,0,0,0,0,0
  DATA 0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0
  DATA 0,1,1,1,1,0,0,0,0,0,1,1,1,1,0,0
  DATA 0,0,0,0,1,0,1,0,1,0,1,0,0,0,0,0
  DATA 0,0,0,1,0,0,1,0,1,0,0,1,0,0,0,0
  DATA 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0
  DATA 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0
  DATA 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0
  DATA 0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0
  DATA 0,0,0,0,1,1,0,0,0,1,1,0,0,0,0,0
  DATA 0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0
  DATA 0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0
  DATA 0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0
  DATA 0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0

  CLS
  FOR y = 0 TO 15
    FOR x = 0 TO 15
      PSET(x+10,y+30), mdata(x,y)
      PSET(x+26,y+30), mdata(x,15-y)
      PSET(x+42,y+30), mdata(y,15-x)
      PSET(x+58,y+30), mdata(15-y,x)
    NEXT x
  NEXT y
  GET (10,30)-(25,45), mouseN
  GET (26,30)-(41,45), mouseS
  GET (42,30)-(57,45), mouseE
  GET (58,30)-(73,45), mouseW

  RETURN

'
' Player moves the mouse through the maze
'
TraverseMaze:
x = xIn
y = yIn
GameOver = false
direction = 1 ' east

CALL MouseMove(0,0,mouseE())

WHILE (NOT GameOver)
  GOSUB GetMove

  IF (move = 4) THEN
    IF (x = 0) OR ((maze(x,y) AND 8) = 0) THEN
      GOSUB Complain
    ELSE
      CALL MouseMove(-1,0,mouseW())
    END IF
  END IF
  IF (move = 3) THEN
    IF ((maze(x,y) AND 2) = 0) THEN
      GOSUB Complain
    ELSE
      CALL MouseMove(1,0,mouseE())
    END IF
  END IF

  IF (move = 2) THEN
    IF (y = yMax) OR ((maze(x,y) AND 4) = 0) THEN
      GOSUB Complain
    ELSE
      CALL MouseMove(0,1,mouseS())
    END IF
  END IF

  IF (move = 1) THEN
    IF (y = 0) OR ((maze(x,y) AND 1) = 0) THEN
      GOSUB Complain
    ELSE
      CALL MouseMove(0, – 1,mouseN())
    END IF
  END IF

  IF (x = xMax + 1) THEN
    GameOver = true
    SAY TRANSLATE$(“nibble, nibble”)
    SAY TRANSLATE$(“Boy, was that tay stee!”)
  END IF

WEND
RETURN

'
' GetMove waits for the user to click on one of the directions
' in the compass
'
GetMove:

  dx = –1
  dy = –1
  WHILE (MOUSE(0) < 0)
    dx = MOUSE(1)
  dy = MOUSE(2)
WEND
```

```
IF dx = -1 THEN GetMove

IF (dx›14) AND (dx‹26) AND (dy›150) AND (dy‹162) THEN
  move = 1
  RETURN
END IF

IF (dx›14) AND (dx‹26) AND (dy›166) AND (dy‹178) THEN
  move = 2
  RETURN
END IF

IF (dx›-1) AND (dx‹11) AND (dy›158) AND (dy‹169) THEN
  move = 4
  RETURN
END IF

IF (dx›29) AND (dx‹41) AND (dy›158) AND (dy‹169) THEN
  move = 3
  RETURN
END IF

GOTO GetMove

'
' MouseMove erases the old mouse, adjusts x and y, and redraws the
' new mouse in the new direction
'
SUB MouseMove(rx,ry,m(1)) STATIC
  SHARED x,y,Csize,TC

  LINE (x*Csize+TC+61,y*Csize+TC+1)-(x*Csize+TC+76,
    y*Csize+TC+16),0,bf
  x = x + rx
  y = y + ry
  PUT (x*Csize+TC+61,y*Csize+TC+1),m
END SUB

'
' See if the player wants to try again
'
AskForAnotherGame:
  CLS
  PRINT
  PRINT " Would you like to play again ";
  INPUT a$
  a$ = LEFT$(a$,1)
  GameWanted = ((a$ = "Y") OR (a$   = "y"))
  RETURN

Complain:
  SAY TRANSLATE$("RATS!")
  RETURN
```

INTERMIXING GRAPHICS AND TEXT

There will be many instances when you want to have both text and graphics on the screen at the same time. To do this requires some understanding about the nature of Amiga text.

In both low and high resolutions, Amiga text characters are all 8 dots high by 8 dots wide. This means that a character in the upper left-hand corner of the screen covers all graphics coordinates from (0,0) to (7,7). Having a fixed character size leads to simple formulas for determining the upper left and lower right graphics coordinates for any character position:

```
upper left coordinate = ((column − 1) * 8,
                         (row − 1) * 8)
lower right coordinate = ((column − 1) * 8 + 7,
                          (row − 1) * 8 + 7)
                                 or
                         ((column * 8) − 1,
                         (row * 8) − 1)
```

where column and row refer to the x and y coordinates of the character's screen position.

However, each Amiga text character is actually only 7 by 7 dots on an 8 by 8 dot field. The extra dot is a border to separate each character from other characters. Furthermore, when a character is on the screen, it completely overwrites everything in its 8 by 8 matrix. Thus, if you want to put graphics flush next to characters, plot the characters first and then draw the graphics.

The following short program makes use of the way the Amiga creates text. The program prints (on the screen) a string of text outlined with a border. You can use the program to highlight certain text, such as a message, or instructions to start a game.

```
DemoBoxPr:
SCREEN 2, 320, 200, 2, 2
WINDOW 2, “Demo BoxPr”,,31,2

CLS
CALL BoxPr(“This is box 1”,1,1,1)
CALL BoxPr(“Press Return      ”,10,10,2)

LOCATE 10,23
INPUT a$

STOP

SUB BoxPr(n$,c,r,kolor) STATIC

  LOCATE c,r
  PRINT n$;

  c1 = (c - 1)*8 - 2           ‘ the - 2 starts the box 2 dots left of the text
  c2 = c1 + LEN(n$)*8 + 11     ‘the +11 ends the box 2 dots right of the text
  r1 = (r - 1)*8 - 2           ‘ the - 2 starts the box 2 dots above the text
  r2 = r1 + 11                 ‘the + 11 ends the box 2 dots below the text

  ‘the + 11’s above are actually + 2 + 7 + 2. The first 2 cancels the effect
  ‘ of the - 2 in the previous equation. The 7 moves to the right-most or
  ‘ bottom-most graphics coordinate in the character and the second + 2 moves
  ‘ two dots beyond that.

  LINE (c1,r1)-(c2,r2),kolor,b        ‘ draw a box around the text

  END SUB

END
```

One other piece of information you may need for mixing graphics and text is the screen position of a character. Again because of the standard character size, the equations for determining the positions are relatively straightforward:

```
character column = INT(xCoordinate / 8) + 1
character row    = INT(yCoordinate / 8) + 1
```

Finally, if you want to create your own text characters, you can draw them as graphic objects, and even animate them. Then you treat the “text” as any other object, moving and positioning it with the Amiga BASIC OBJECT statements.

Part III

SOUND ON THE AMIGA

CHAPTER 7

Creating Sound

Sound is the Amiga's forte. By programming its four-voice sound capabilities, you can produce electronic music approaching the quality of commercial synthesizers and realistic sounds for games. In addition, the Amiga has a set of software routines built into ROM that create speech synthesis. A complete range of Amiga BASIC commands are specifically designed to take advantage of these sound features. Knowing which BASIC command to use depends on the type of sound you want and how you want to create it. That, in turn, means you need to know a little about how sound is produced by the Amiga.

UNDERSTANDING AMIGA SOUND

Sound waves, as they reach your ears, are the vibrations of air. The faster the air vibrates, the higher the pitch of the sound, and the harder it vibrates, the louder the sound. In terms of the wave itself, the frequency of the wave is what makes the air vibrate at a certain speed and thus determines the pitch, while the amplitude, or height, determines the volume (Figure 7-1).

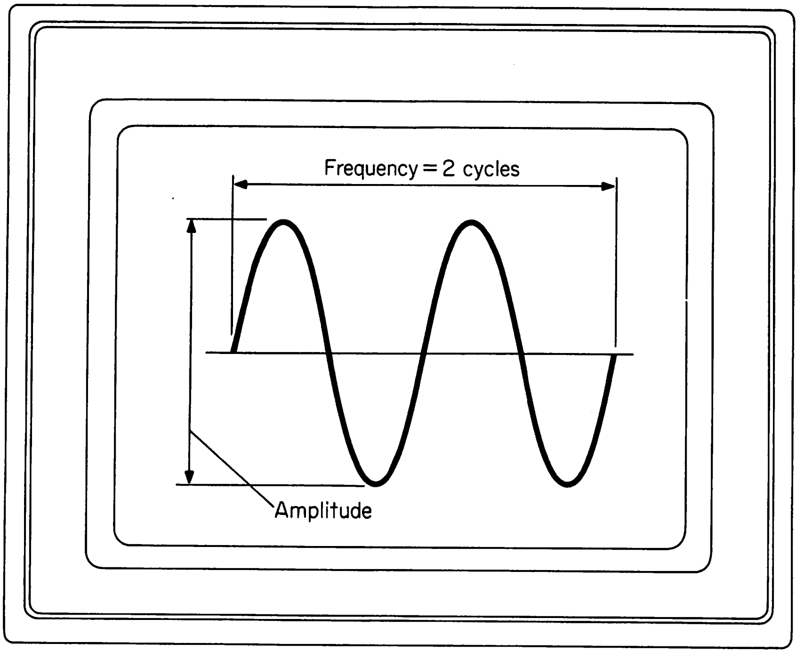


Figure 7-1

Simple sound waves, such as the sine wave, have clear, pure tones. The Amiga has a special command to produce sine-wave sounds. Musical instruments, on the other hand, produce far more complex waveshapes, and it is this complexity that gives each instrument its typical sound and explains why, for instance, a banjo playing middle C sounds different from a trumpet playing exactly the same note.

Complex waveshapes have harmonics, which are mathematically exact multiples of the main frequency of an instrument's note. The amount of each harmonic present in a wave-shape determines the note's timbre. A sawtooth wave with harmonics produces sound that is "brassy," while a triangular wave sounds more like a clarinet. Thus, an instrument's notes sound relatively the same regardless of the frequency of the note.

Audible frequencies are from about 20 to about 20,000 hertz (Hz), or cycles per second. In musical instruments, different notes are produced when the musician physically "alters" the way the sound is made — by sliding a button up or down on the trumpet, or by pressing on the banjo string, effectively shortening how much of it vibrates. Volume depends on how hard the trumpeter blows into the mouthpiece or how hard the banjo picker plucks the string.

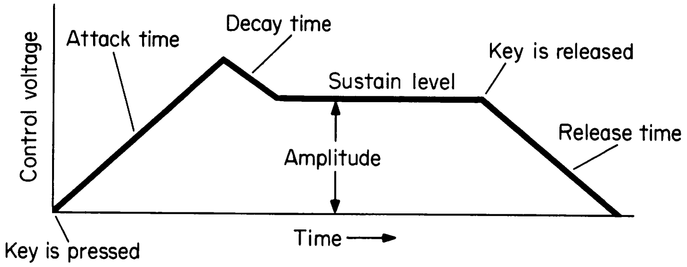
In the vocabulary of electronics, all of the processes that instruments use to produce sound are known as *analog* processes. That is, they depend on physical relationships, like length of string or vibrating frequency, to produce sound. For computers to produce sound, they must somehow duplicate these analog processes in a digital manner. Because computers work on electricity, the analog process they use is voltage.

Computers use a digital-to-analog (or D-to-A) converter to produce sound. It works like this: a D-to-A converter is an electrical device that sends voltage to the stereo sound speakers attached to your Amiga. As the voltage fluctuates, the sound emitted by the speakers changes. The computer controls the voltage output of the converter by sending it digital information in the form of byte values. Thus, as the byte values (the digital information) change, the voltage (the analog information) changes, and different sounds come from your speakers.

The Amiga can produce four separate voices simultaneously. The output voices are numbered 0, 1, 2, 3. With stereo speakers connected to the computer, voices 0 and 3 go to the left speaker, and 1 and 2 go to the right speaker.

Inside the Amiga, each voice has its own DMA channel to the 68000. The digital information on each channel includes volume, waveshape, and how often the byte data must be sent to the D-to-A converter. This information creates a sound table, which software in the ROM then uses to create an *envelope* for the waveshape. Sound envelopes for electronic instruments, including synthesizers as well as computers, describe the voltages sent to the D-to-A converter. The shape of an envelope replicates the way a sound actually occurs. Consider, for instance, a piano key. When you first press a key, the sound is sharp and clear, but then fades slightly, followed by a period when the sound is still clear but at a lower level. When you release the key, the sound fades out. A sound envelope duplicates that sequence of events with four features (Figure 7-2):

Attack — time for sound to reach its peak level
Decay — time for sound to reach its sustaining level



Typical ADSR

Figure 7-2

Sustain — sound of the key as it is held down
Release — time after key is released until sound fades

In some computers you must define each element of the attack-sustain-decay-release (ASDR) curve in order to produce sound. Such is not the case with the Amiga. Amiga BASIC's commands include those to produce the notes for you as well as those that require an understanding of ASDR.

PROGRAMMING SOUND

The easiest BASIC command for programming sound is BEEP. That command creates a short beep sound from the Amiga's internal speaker and simultaneously flashes the screen. You use this command to get the attention of the person sitting in front of the computer. The SOUND and WAVE commands in Amiga BASIC are the ones you use to create music or sound effects. SOUND does not require any type of ASDR curve; WAVE requires an array of numbers that define a waveform. A BASIC program in chapter 9 creates wave curves for you, which simplifies programming with WAVE.

If you have programmed with other versions of BASIC, you may be familiar with a command PLAY. PLAY is for writing music with many of the traditional notations and conventions now used by professional composers and arrangers. The current version of Amiga BASIC does not include a PLAY command, but a BASIC program in chapter 8 of this book provides the PLAY command for you. You can add it to your Amiga BASIC disk and treat it as a regular BASIC command.

Finally, two commands to create electronic speech are SAY and TRANSLATE$.

Using the SOUND Command

To program music using the SOUND statement, you tell the computer the frequency of the note to be played, the note's duration, volume, and voice. A typical SOUND statement is:

```
SOUND 440, 10, 100, 1
```

The number 440 refers to the frequency of the note to be played. Frequency is expressed in hertz. The frequency 440 is known as the "tuning A," because it is the frequency of the piano key that tuners use as a reference tone for tuning the rest of the keys. Other frequencies on either side of tuning A are:

Note	Frequency
C	130.8
D	146.8
E	164.8
F	174.6
G	196.0
A	220.0
B	246.9
C	261.6 (middle C)
D	293.6
E	329.6
F	349.2
G	392.0
A	440.0
B	493.8
C	523.2
D	587.3

E	659.2
F	698.4
G	783.9
A	880.0
B	987.7
C	1046.5
D	1174.7
E	1318.5
F	1396.9
G	1568.0
A	1760.0
B	1975.5

To save yourself time, you can ignore the last number in each of the frequencies. Most people can't tell the difference of a note within 3 to 5 Hz.

There is a discrepancy between the middle C listed in some books on electronic music and the middle C frequency quoted by professional musicians. Middle C to musicians is 261.6, not the 523.25 in the Amiga BASIC manual. This difference can sometimes create problems when transcribing sheet music into electronic form.

Each group of eight notes in the table is an octave. Sharps and flats are not shown, but finding their frequency values is easy. A sharp is a half step up in frequency from a note, while a flat is down a half step from a note. To find a sharp or flat, multiply the note by 1.06 (or if you're a purist, by 1.059463). For instance, to find C sharp (denoted as C#) above middle C, multiply 261.6 by 1.06 to get 277.3 Hz.

To program the computer at frequencies either higher or lower than shown on the table, double or halve the frequencies of the notes in the preceding or following octave. For instance, the first C in the table is 130.8 and the next C is 261.6 (middle C), about double the frequency of the first one. Similarly, the last A in the table is 1,760.0, so the next octave's A (not shown on the table) would be about 3,520.0. The Amiga can play frequencies from 20 to 15,000 Hz.

The second number in the SOUND statement, 10, sets the duration of the note. Duration is measured in clock "ticks" of the Amiga's internal clock, which ticks 18.2 times per second, or 1,092 ticks per minute. Duration in the SOUND statement can be any number from 0 to 77. A note programmed for the full 77 would play for 4.25 seconds.

The duration is for setting the *tempo* of a piece of electronic music. Tempo is the speed at which the piece is played. The lower the number, the shorter the duration of the notes and the faster the music. Very slow music, larghetto, that plays at about 60 to 66 beats per minute, would have a duration of 18.2 to 16.55 ticks per beat. Presto, or very fast music, played at 168 to 208 beats per minute would have durations of 6.5 to 5.25 ticks per beat. Moderately fast music has a duration of about 100. The duration numbers for a particular piece can be calculated by dividing the music's number of beats per minute by 1,092, the number of clock ticks per minute.

The third number in the SOUND statement sets the volume of the note. In the example, the number 100 produces a moderately soft note. Volume can range from 0 to 255 which increases the sound into successive, louder output. Volume does not have to be specified in the SOUND statement. If it isn't, the Amiga automatically plays the note at a volume of 127.

The last number in the SOUND statement assigns the music to one of the four voices. The numbers 0 and 3 assign the output to the left speaker; 1 and 2 put the sound through the right speaker. If you don't specify the voice, the Amiga automatically assigns it to 0, the left speaker.

Programming the Tune "Happy Birthday" Using the SOUND Statement

Here's a BASIC program using the most elementary steps for creating computer music. First, all the notes used in the melody need to be translated into the necessary frequencies and durations required by the SOUND statement. The different notes of "Happy Birthday" are G, A, O2C (where the O2 signifies that the note is played in the next octave up), B, O2D, O2G, O2E, and O2F. The three different kinds of notes in the piece are 1/8, 1/4, and 1/2 notes. Thus, the data you need for this program is:

Note	Frequency
G	392.0
A	440.0
B	493.8
O2C	523.2
O2D	587.3
O2E	659.2
O2F	698.4
O2G	783.9

Kind of Note	Duration in Amiga clock ticks
1/8	4.55
1/4	9.1
1/2	18.2

One of the problems of transcribing sheet music to computer program format is keeping track of the notes in their proper positions. The following "translation table" is a method of organizing the notes that makes transcribing them to computer format fast and simple (Figure 7-3).

In musical terms, the notes that make the melody for "Happy Birthday" are:

Note	Kind of note
G	1/8
G	1/8
A	1/4
G	1/4
O2C	1/4
B	1/2
G	1/8
G	1/8
A	1/4
G	1/4
O2D	1/4
O2C	1/2
G	1/8
G	1/8
O2G	1/4
O2E	1/4
O2C	1/8
O2C	1/8
B	1/4
A	1/4
O2F	1/8
O2F	1/8
O2E	1/4
O2C	1/4
O2D	1/4
O2C	1/2

A SIMPLE TRANSLATION TABLE
FOR HAPPY BIRTHDAY

NOTE LENGTHS	8 8	4 4 4 2 8 8	4 4 4 2 8 8	4 4 8 8	4 4 8 8	4 4 4 2

NOTE	G G	A G C B G G	A G D C G G	G E C C	B A F F	E C D C
OCTAVE	1 1	1 1 2 1 1 1	1 1 2 2 1 1	2 2 2 2	1 1 2 2	2 2 2 2

Figure 7-3

Using the two tables, you can now program the tune with the SOUND statement. The first number in each of the following SOUND statements is the frequency of the note and the second number is the duration. In the interest of saving you some typing time, frequencies with only a zero after their decimal points are typed as three-digit numbers. (You could also type all of the frequencies as three-digit numbers and for all intents and purposes the music would sound the same.) Observe that you do not have to specify the volume or voice. Those default settings are 127 and 0.

```
SOUND 392, 4.55
SOUND 392, 4.55
SOUND 440, 9.1
SOUND 392, 9.1
SOUND 523.2, 9.1
SOUND 493.8, 18.2
SOUND 392, 4.55
SOUND 392, 4.55
SOUND 440, 9.1
SOUND 392, 9.1
SOUND 587.3, 9.1
SOUND 523.2, 18.2
SOUND 392, 4.55
SOUND 392, 4.55
SOUND 783.9, 9.1
SOUND 659.2, 9.1
SOUND 523.2, 4.55
SOUND 523.2, 4.55
SOUND 493.8, 9.1
SOUND 440, 9.1
SOUND 698.4, 4.55
SOUND 698.4, 4.55
SOUND 659.2, 9.1
SOUND 523.2, 9.1
SOUND 587.3, 9.1
SOUND 523.2, 18.2
RUN
```

Different programming can reduce the list of SOUND statements for a song, but at some point you always have to list each note frequency with its duration. Using the PLAY statement described in chapter 8 makes programming music much easier.

PROGRAMMING SPEECH

The Amiga is a machine of many words. The secret of its speech is found in an Amiga program on the Workbench disk, called Narrator Device. That program contains a set of rules and table of pronunciation for the English language. When you enter a phrase for the Amiga to speak, the program searches through its rules and table to find the appropriate sounds, and then sends the result to the stereo speakers.

The two Amiga BASIC commands for speech synthesis are SAY and TRANSLATE$. The SAY command has two functions: first it tells the computer that you want it to speak, and second it can describe *phonemes* of speech. Phonemes are to speech what atoms are to matter — they are the smallest building blocks of speech. For example, in the words *pat* and *fat*, the *p* and *f* are two different phonemes in English. You can use the SAY command by itself to create speech, but it requires breaking down each syllable of sound into individual phonemes.

The TRANSLATE$ command does not require the phonemes. TRANSLATE$ tells the Amiga what you want to say in English, or if you spell foreign words in phonetic English, the TRANSLATE$ command can also make the Amiga speak in a foreign language (although with an English accent!).

The easiest way to produce speech is with a one-line statement:

```
SAY TRANSLATE$("Good Morning.")
```

The first time you type a SAY statement, the Amiga must load the Narrator Device program from the Workbench disk, so be sure it's in the drive. The TRANSLATE$ command translates into English the phrase enclosed in parentheses and quotes. Everyday words are no problem for the TRANSLATE$ command; ambiguous symbols or phrases, however, get treated specially. Numbers, for instance, are translated individually so that the number 67 is spoken as "six seven," not "sixty-seven." As the following table shows, other symbols have their own translations as well.

Symbol	What the Amiga Says
>	Greater than
<	Less than
=	Equals
+	Plus
.	Point (only if it precedes a number; otherwise treated as a period)
#	Number
$	Dollar
%	Percent
"	Quote (first time it occurs), unquote (second occurrence), quote (third occurrence, and so on)
&	And
¦	Or
...	And so on
/	Slash
^	Caret
~	Tilde

Notice that the minus sign (−), multiplication sign (x), and the division sign (/) are not represented in the table. Amiga treats the minus as a dash or hyphen for a pause in the speech. The multiplication sign is pronounced as the letter *x*, and the division sign is slash. You must type out the words "minus," "multiply by," and "divide by" for them to be spoken.

Punctuation marks are not pronounced, but can affect the spoken inflection. A period or question mark at the end of a sentence causes a brief pause and a drop in the pitch of the speech. No punctuation mark at the end of a sentence raises the pitch to give it an interrogative inflection. The comma, colon, and semicolon cause appropriate pauses.

A hint for intelligible speech: keep sentences short or use punctuation marks to break them up with pauses. Because the Amiga never has to catch its breath, long or unbroken sentences sound unnatural.

The Amiga's speaking voice (not to be confused with its four voices for music or sound) can take on different qualities. You define the qualities in a one-dimensional array consisting of nine elements for the SAY statement. If you define the array with the DIM statement as speech(9), the format for the array in the SAY TRANSLATE$ statements is:

```
SAY TRANSLATE$ ("The text you want to say"), speech%
```

The nine elements of the array correspond to:

1 pitch of speech
2 inflection
3 rate of speech
4 gender of voice
5 tuning, which affects pitch and "quality of the speech" such as rumbly or squeaky
6 volume
7 speaker output
8 delay for other programs
9 interrupts or cancels other speech

See the Amiga BASIC manual for the optional values for each of these elements. For instance, element number 3, gender, has two options: 0 for male and 1 for female. The best way to decide on the voice you want to give to your Amiga is to try out different combinations of the array.

Speaking Phonetically

Because of English's quirks, pronunciation is a subtle and complex job. Although people learn correct pronunciation around the age of five, the computer occasionally still needs some help. One way to tell the Amiga the correct pronunciation of a word is to write it out in phonemes (as described later). A simpler way is to deliberately misspell a word, or to put it more politely to spell phonetically. Longer words are especially receptive to phonetic spelling. As with determining the best speaking voice for the Amiga, determining the best spelling for words you want to use is a trial-and-error process. When you find useful phonetic spellings, write them down and develop your own "Amiga words dictionary."

Phonetic spelling is also one way to program the Amiga to speak in a foreign language. For instance, the German phrase for good morning, *guten morgen*, becomes "goo ten more gen." Generally you can phonetically spell foreign words that have English equivalent sounds, but not those that use various clicking or hissing noises as a part of speech. Furthermore, some languages having English-equivalent sounds also have peculiar characteristics that require special treatment. French, with its heavy nasal emphasis, is a prime example.

You can also phonetically spell words to give them regional accents. Spelling the word "yawl" for *you all*, and "heeyah" for *hear* puts a distinctly southern emphasis on the phrase, "Yawl kum bak an see me now, heeyah?"Similarly, the phrase Lawn Guyland is familiar to any Noo Yawker living on Long Island.

To try different phonetic spellings of words, use the following short program. While the program is running, you can type words on the Output screen without writing the SAY or TRANSLATE$ statements. As soon as you press the Return key at the end of each phonetically spelled phrase, the Amiga speaks.

```
LOOPTOP:
LINE INPUT A$
PRINT TRANSLATE$(A$)
SAY TRANSLATE$(A$)
GOTO LOOPTOP
RUN
```

Using Phonemes

As discussed earlier, phonemes are the elemental sounds of a language. In dictionaries, the symbols of pronunciation, such as an *e* with a bar over it denoting a long *e*, are representations of phonemes. You can program speech in phonemes using the SAY statement by itself. A typical SAY statement programmed with phonemes looks like this:

```
SAY "DHIHS IHZ NAATQ TUW /HAORD TUW
        AH1NDERSTAE4ND"
```

which says, "This is not too hard to understand."

The following table shows how Amiga BASIC phonemes correspond to the pronunciation symbols in a *Webster's New Collegiate Dictionary*.

You can program the phrases roughly using these symbols as a guide; however, you'll probably find a few words that require some modification to sound right.

Symbol	Example	Amiga BASIC
∂	abort, banana, collide	AX
′∂,ˏ∂	under, humdrum, abut	AH
(∂)	solid, sour, wire	IX
∂r	further, bird, meager	ER
a	bat, map, gag	AE
ā	made, date, drape	EY
ä	hat, bother, cart	AA
ȧ	talk, palm, tomato	AO
au̇	power, loud, out	AW
b	but, baby, rib	B
ch	chin, check, nature	CH
d	dog, did, adder	D
e	bet, bed, peck	EH
′ē,	beat, evenly, nosebleed	IY
f	fed, fifty, cuff	F
g	go, guest, big	G
h	hole, hat, ahead	/H
i	bit, tip, active	IH
ī	hide, buy, site	AY
j	judge, job, gem	J
k	kin, Commodore, ache	K
k̲	loch, (German ich, buch)	/C
l	lily, yellow, pool	L
m	men, murmur, dim	M
n	no, own, men	N
η	sing, finger, ink	NX
ō	low, bone, oboe	OW
ȯ	border, saw, caught	OH
ȯi	boil, coin, destroy	OY
p	put, pepper, lip	P
r	red, rarity, beard	R
s	sail, source, less	S
sh	shy, rush, machine	SH

Symbol	Example	Amiga BASIC
t	toy, late, latter	T
th	thin, ether, thigh	TH
t̲h̲	then, either, this	DH
ü	crew, rule, youth	UW
u̇	look, pull, wood	UH
v	very, vivid, invite	V
w	we, away, wend	W
y	yellow, yard, young	Y
z	zone, raise, has	Z
zh	pleasure, vision, azure	ZH

Other symbols for the SAY statement include numbers that indicate stress marks, and a series of letter combinations for special sounds. See Appendix H of the Amiga BASIC manual for a complete description of the phonemes.

The benefit of programming with phonemes is that it gives you the most flexibility and control over the speech. You can control emphasis, inflection, pronunciation, pacing, pauses, and other features of speech. And foreign languages lose some of their English accent when programmed in phonemes. The disadvantages are that it takes longer than using the easy TRANSLATE$ statement, and it requires becoming intimately familiar with phonemes and complex word constructions. Also, the SAY command takes some getting used to. It only accepts uppercase letters and treats some phoneme combinations as illegal.

A final note about speech synthesis: computer-generated speech is a growing field. Synthesized speech is prevalent in elevators, telephone answering systems, automobile warning systems, vending machines, and so on. Knowing the principles of speech synthesis may stimulate you to find a new application for this interesting use of computers.

CHAPTER 8

Synthesized Music

With its four-voice sound and ability to attach a MIDI (Musical Instrument Digital Interface) adaptor for connecting synthesizers to it, the Amiga has a promising future in music. You can also program the Amiga to compose or arrange your own tunes. Composing music means making up the tunes yourself starting from scratch. Arranging music means taking the songs of other composers and making the music sound like you want it. For instance, giving a Bach piece a rock-and-roll beat would be rearranging it. The songs in this chapter are arrangements of existing music.

Transcribing a song to the Amiga may require special arranging to take advantage of the computer's many musical talents, and to reconstruct a tune with the four voices. Some songs sound best with all four voices, while others only need two or three. And as you'll see in the next chapter, the arrangements can be for different instruments for the different Amiga voices.

To facilitate arranging and composing, the following program produces a BASIC statement known as PLAY. Found in other versions of BASIC, PLAY is more powerful and faster than the SOUND statement. Copy this program onto a disk. To use it to create music, load Amiga BASIC and load this program. You then type the music programs and insert them in the space in the program after the REM statement "PUT YOUR CALL PLAY AND CALL MP STATEMENTS HERE." The programs for "Happy Birthday" and "Silent Night" are there to illustrate how you use the program. Erase those two tunes before you enter your new songs.

THE PLAY PROGRAM

```
REM Program to demonstrate the PLAY subprogram
REM Written by Joseph R Power

InitPlay = 1

REM This is where the programs for assigning
REM waveforms to the different voices will go

REM PUT YOUR CALL PLAY AND CALL MP STATEMENTS HERE

REM Remember to erase the example songs of
REM Happy Birthday and Silent Night before
REM adding your new song to the program

REM CALL mp("ABCD","AACC","DDDD","GFED")
REM This is Happy Birthday
CALL PLAY("G8 G8 A4 G4 › C4 ‹ B2",1)
CALL PLAY("G8 G8 A4 G4 › D4 C2 ‹",1)
CALL PLAY("G8 G8 › G4 E4 C8 C8 ‹",1)
CALL PLAY("B4 A4 › F8 F8 E4 C4 D4 C2",1)

REM This is Silent Night
CALL PLAY("T60O2",1)
CALL PLAY("V2F8.G16F8D4.F8.G16F8D4.›C4C8‹A4.B-4B-8F4.",1)
CALL PLAY("G4G8B-8.A16G8F8.G16F8D4P8G4G8B-8.A16G8",1)
CALL PLAY("V15F8.G16F8D4P8‹C4C8E-8.C16‹A8B-4.›D4.‹B-8.",1)
CALL PLAY("F16D8F8.E-16C8‹B-2P8›E-2D2",1)
END

SUB mp(v0$,v1$,v2$,v3$) STATIC
SOUND WAIT
IF v0$ ‹› "" THEN CALL PLAY(v0$,0)
IF v1$ ‹› "" THEN CALL PLAY(v1$,1)
IF v2$ ‹› "" THEN CALL PLAY(v2$,2)
IF v3$ ‹› "" THEN CALL PLAY(v3$,3)
```

```
SOUND RESUME
END SUB

SUB PLAY(P$,Voice%) STATIC
SHARED InitPlay

IF InitPlay = 0 THEN StartPlay

DIM NVals(27), N2Vals(12), PZ%(256)

FOR i = 0 TO 256
  PZ%(i) = 0
NEXT i

FOR i = 1 TO 27
  READ NVals(i)
NEXT i
DATA 131, 139, 0, 147, 139, 0, 156, 0
DATA 165, 156, 0, 175, 185, 0, 196, 185, 0, 208, 0
DATA 220, 208, 0, 233, 0, 247, 233, 0

FOR i = 1 TO 12
  READ N2Vals(i)
NEXT i
DATA
131,139,147,156,165,175,185,196,208,220,233,247

Tempo = 34.125
Volume = 128
Octave = 1
NLen = 1
InitPlay = 0
StartPlay:
  i = 1
  P = LEN(P$)
parse:
GOSUB NextNote
  IF N$ = “” THEN PRINT: EXIT SUB
  N = INSTR(“ABCDEFGONLPT›‹V”,N$)
  IF N = 0 THEN GOTO parse

  ON N GOSUB A, B, C, D, E, F, G, O, N, L, P, T,
GT, LT, V
  GOTO parse
A:
```

```
B:
C:
D:
E:
F:
G:

  s$ = N$
  REM
  REM Check for sharps and flats
  REM
  GOSUB NextNote
  IF N$ = “#” THEN N$ = “+”
  IF (N$ = “+”) OR (N$ = “−”) THEN s$ = s$ + N$
ELSE i = i − 1
  REM
  REM Check for a single note duration
  REM
  GOSUB GetNum
  IF NumVal = 0 THEN duration = NLen ELSE duration
= NumVal
  REM
  REM Check for dotted notes
  REM
  GOSUB NextNote
  IF N$ = “.” THEN duration = duration * 1.5 ELSE i
= i − 1
  REM
  REM Get note's frequency from table
  REM
  Note =
NVals(INSTR(“CC+DD−D+EE−FF+GG−G+AA−A+BB−”,s$))
  SOUND
Note*(2^Octave),Tempo/duration,Volume,Voice%
  FOR z = 1 TO 5: x = SIN(10): NEXT z
  RETURN
O:
  GOSUB GetNum
  IF NumVal › 6 THEN Octave = 6 ELSE Octave =
NumVal
  Octave = Octave − 2
  RETURN
N:
  GOSUB GetNum
  IF NumVal › 84 THEN Note = 0 ELSE Note = NumVal
  IF Note = 0 THEN WAVE Voice%,PZ%
  IF Note = 0 THEN Vol = 0 ELSE Volume
  O = 1 + INT((Note - 1)/12)
  Note = Note MOD 12
  IF Note = 0 THEN Note = 12
  Note = N2Vals(Note)
  SOUND Note * (2^0), Tempo/NLen, Vol, Voice%
  WAVE Voice%,SIN
L:
  GOSUB GetNum
  IF (NumVal = 0) OR (NumVal › 64) THEN NumVal = 1
  NLen = NumVal
  RETURN
P:
  GOSUB GetNum
  IF (NumVal = 0) OR (NumVal › 64) THEN NumVal = 1
  WAVE Voice%,PZ%
  SOUND 20,Tempo/NumVal,0,Voice%
  WAVE Voice%,SIN
  RETURN
T:
  GOSUB GetNum
  IF (NumVal ‹ 32) OR (NumVal › 255) THEN NumVal =
120
  Tempo = 1092/NumVal * 4
  IF Tempo › 77 THEN Tempo = 77
  RETURN
GT:
  GOSUB GetNum
  IF NumVal = 0 THEN Octave = Octave + 1 ELSE
Octave = Octave + NumVal
  IF Octave › 4 THEN Octave = 4
  RETURN
LT:
  GOSUB GetNum
  IF NumVal = 0 THEN Octave = Octave - 1 ELSE
Octave = Octave - NumVal
  IF Octave ‹ -2 THEN Octave = -2
  RETURN
V:
  GOSUB GetNum
  Volume = NumVal * 16
  RETURN
NextNote:
  IF i › P THEN N$ = “” ELSE N$ = MID$(P$,i,1)
  i = i + 1
  RETURN
GetNum:
  NumVal = 0
GN1:
  q = INSTR("0123456789",MID$(P$,i,1)) − 1
  IF q = −1 THEN GN2
  NumVal = NumVal * 10 + q
  i = i + 1
  IF i ‹= P THEN GN1

GN2:
  RETURN
END SUB
```

This program is essentially a parser that looks at each PLAY statement and breaks down notes into their discrete values. Then the program matches the notes to their frequencies, octaves, and voices.

Although the remarks throughout the program explain what each section does, you have to know a bit about the Amiga to understand what the different statements accomplish. For example, the statement for tempo is tempo = 1092/NumVal * 4. Tempo to the PLAY statement is the number of quarter notes in a minute. The Amiga's internal clock "ticks" 1,092 times per minute and the NumVal is the number you assign to the tempo. Therefore, multiplying the ratio of 1,092 and NumVal by 4 determines the quarter notes per minute for your tempo setting.

PROGRAMMING WITH THE PLAY STATEMENT

The PLAY statement recognizes notes using their standard names and musical notations. A typical PLAY statement is

```
PLAY ("G8G8A4G4›C4‹B2",1)
```

which is the first six notes of the "Happy Birthday" melody. The first two G8's are the PLAY notation for playing two successive G 1/8 notes, A4 is for an A 1/4 note, and G4 is for a G 1/4 note. The › makes the next note, a C 1/4 note, play up one octave (equivalent to the O2C note in the SOUND tables), and the ‹ brings the octave back down for the next note, the B 1/2.

The 1 at the end of the notes assigns the output for the music to the Amiga's 1 channel of its four voices. Continuing the programming, here is the entire "Happy Birthday" melody that you programmed in chapter 7 done instead in one simple PLAY statement:

```
PLAY (“G8G8A4G4›C4‹B2G8G8A4G4›D4C2‹G8G8›G4E4
     C8C8‹B4A4›F8F8E4C4D4C2”,1)
```

Note that all the notes must be enclosed in parentheses and quotes, and that the voice designator must always be at the end of the notes.

A Reminder

Remember to precede the PLAY statement with a CALL command when you type the music into the PLAY program (as illustrated by the examples). In other words, when you enter the Happy Birthday PLAY statement into the program it will be CALL PLAY (“G8G8A4G4›C4...and so on). To avoid repetition, the rest of the music programs in this chapter do not include the CALL statement, but remember to include them with your music in the PLAY program.

The PLAY statement also prepares the computer for the type of music it is to play. For example, before writing the PLAY statement with the notes for the melody, you could write:

```
PLAY (“T100O2”,3)
```

This PLAY statement tells the Amiga to play the music at a tempo of 100 beats per minute (T100), to start the music in octave 2 (O2), and to play the music through voice channel 3. The *O* in the O2 for octaves is a capital letter *O* and not the number 0. The › and ‹ symbols in the first PLAY statement are relative to the octave in which the piece was started.

Some other notations for the PLAY statement are:

Names of notes — A,B,C,D,E,F,G.

Sharps and flats — # or + after a note indicates it is a sharp; a − (minus sign) after a note means it is a flat. Sharps and flats can only indicate the black keys on a piano. Invalid notes in computer music are B + , C − , E + , and F − .

Kind of notes — 1,2,3,4,5,6,7,8,..., 64. A number 1 is a whole note, 2 is a half note, 3 is one of a triplet of three half notes (1/3 of a 4-beat measure), 4 is a quarter note, 5 is one note of a quintuplet (1/5 of a measure), and so on.

Dotted notes — a dot or period after a note makes the Amiga play the note as a dotted note. A dotted note is played 1/2 again as long as it normally would. Some versions of the PLAY command allow double dotted notes (two periods following the note), but this version only allows single dotted notes. The dots must follow after the symbols that name the note and its kind; e.g., F + 4. is the right form, F + .4 is not. (If you're interested in adding the double dotting feature to the PLAY program, double dotted notes play a dotted note plus 1/2 of the duration added by the first dot. Mathematically this turns out to be 7/4 as long as the original note.)

Octave — an uppercase letter *O* followed by a number selects the octave for a note or a melody; › and ‹ increase and decrease the octave by one for the note following the symbols; › and ‹ followed by a number would increase or decrease the next note in a tune by that number of octaves (for instance, ‹3 would decrease the next note in a tune by 3 octaves). If you do not specify an octave for a tune, the Amiga plays it at octave O1 which is the octave containing middle C. The Amiga can play seven octaves, numbered 0 to 6. Octave 0's lowest note is C which corresponds to a frequency of 131 Hz.

Pause — a *P* indicates a rest, i.e., no sound. The same numbers used to indicate note lengths designate lengths of rests.

Tempo — the letter *T* followed by a number between 32 and 255 sets the tempo of the music. If tempo is not specified, the Amiga plays the music at a tempo of 120 which is equivalent to music played *Moderato* (medium fast). The tempo number itself defines the number of quarter notes that would be played in one minute. The higher the number, the faster the music is played.

Volume — a *V* followed by a number between 0 and 15 sets the volume of the note or notes after the V number: 0 is softest, 15 is loudest.

Although some of these notations are written in separate PLAY statements, such as the tempo and the octave of a tune's first note, they can also be written as part of the PLAY statement that contains the notes themselves. Putting certain notations in separate PLAY statements merely makes it easier to see what type of music is being played. That convention is adhered to in the following music examples.

Programming the Melody for "Silent Night"

The popular Christmas carol, "Silent Night," is played slowly, as a hymn. It was originally composed in German as a guitar accompaniment ("Stille Nacht") to replace standard religious hymns when a church organ broke down just before Christmas.

An Amiga program for the melody of "Silent Night" is:

```
PLAY ("T60O2",1)
PLAY ("F8.G16F8D4.F8.G16F8D4.›C4C8‹A4.B – 4B – 8
    F4.G4G8B-8.A16G8F8.G16F8D4P8G4G8B – 8.A16G8
    F8.G16F8D4P8›C4C8E – 8.C16‹A8B – 4.›D4.‹B – 8.
    F16D8F8.E – 16C8‹B – 2P8›E – 2D2",1)
RUN
```

The first line of the program defines the music's tempo (T60) at a slow 60 quarter notes per minute (about the right speed for a hymn) and starts the music at octave 2 (O2). The second line lists the actual notes of the melody of "Silent Night." "Silent Night" begins with F dotted 1/8, G 1/16, F 1/8, D dotted 1/4, and F dotted 1/8. Following an 1/8 pause (P8) near the end of the line, the last two notes (E-2D2) in the program are a refrain for "Amen."

The notes could also be written with a space between them, such as PLAY ("F8. G16 F8 D4. ...and so on). The spaces do not affect the way the tune is played and make the notes easier to read.

Harmony and Accompaniment

The PLAY statement lets the Amiga take full advantage of any one of the four voices of its sound generator. Another statement (which is part of the PLAY program) is Multiple Play or MP for short. You use MP whenever you want the music to be played with more than one Amiga voice.

As with virtually all BASIC programming, creating a music program that plays harmony can be done a number of different ways. The method about to be described represents one simple, effective way of getting the Amiga to play a harmonized melody for "Silent Night." The arrangement uses all four voices.

The first line describes the tempos and octaves of each of the four voices. Each set of letters within the quote marks is for one of the voices. The position of the descriptions in the line determine which voice it is. That is, the first description is for voice 0, the second one is for voice 1, the third for voice 2, and the last one for voice 3. The MP command always requires descriptions for four voices even if the musical piece uses less than all four. In that case the description for the unused voice is a blank enclosed by quotes, i.e., " ".

In this tune, voice 0 is to be played at a tempo of 60 (T60) and is to start at octave 2 (O2); voice 1 is to be played at the same tempo and octave as the first voice. But voice 2 is to be played at a tempo of T60 at one octave below (O1) the first and second voices. Voice 4 begins at the lowest octave (O0).

The next lines of the program differ from earlier examples of the PLAY statement. Rather than telling the Amiga to just play a list of notes, the following lines define the music notes as variables. The V1$, V2$, V3$, and V4$ in the program indicate that the data (in this case, notes) that follows is a variable. Although not necessary, the four statements are called V1 to indicate voice 1, V2 for voice 2, and so on. The statements could just as easily have been called A$, B$, C$, and D$, but naming them with the *V*'s helps you remember which line refers to which Amiga voice.

As you can see, the first voice's notes (after the V1$) are merely a repeat of the melody written earlier. The notes after the V2$, V3$, and V4$ are for the harmony.

```
MP ("T60O2","T60O2","T60O1","T60O0")

V1$= ("F8.G16F8D4.F8.G16F8D4.›C4C8‹A4.B-4B-8F4.
        G4G8B-8.A16G8F8.G16F8D4P8G4G8B-8.A16G8
        F8.G16F8D4P8›C4C8E-8.C16‹A8B-4.›D4.P8‹B-8.
        F16D8F8.E-16C8‹B-2P8›E-2D2",0)

V2$= ("D8.E-16D8‹B-4.›D8.E-16D8‹B-4.›E-4E-8E-4.
        D4D8D4.E-4E-8E-8.E-16E-8D8.E-16D8‹B- 4P8›
        E-4E-8E-8.E-16E-8D8.E-16D8‹B-4P8›E-4E-8
        A8.E-16E-8D4.F4.P8D8.D16‹B-8›D8.C16‹B-8F2
        P8B-2B-2",1)

V3$= ("B-8.B-16B-8F4.B-8.B-16B-8F4.A4A8›C4.‹F4
        F8B-4.B-4B-8B-8.›C16‹B-8B-8.B-16B-8F4P8
        B-4B-8B-8.›C16‹B-8B-8.B-16B-8F4P8A4A8›C8.‹
        A16›C8‹F4.B-4.P8F8.B-16F8F8.F16F8D2
        P8G2F2",2)

V4$= ("B-8.B-16B-8B-4.B-8.B-16B-8B-4.›F4F8F4.‹
        B-4B-8B-4.›E-4E-8G8.F16E-8‹B-8.B-16B-8
        B-4P8›E-4E- 8G8.F16E-8‹B-8.B-16B-8B-4
        P8›F4F8F8.F16F8‹B-4.B-4.P8B-8.B-16B-8›
        F8.F16E-8‹B-2P8B-2B-2",3)

MP (V1$,V2$,V3$,V4$)
RUN
```

If you copy this program, be sure to type it exactly as shown, including the quote marks and equal signs. Insert it at the proper place in the PLAY program (as indicated by the REM statements). Once you've heard it played, you can add minor improvements to the music by changing some of the variables. For example, if a tempo of 60 is too slow for your taste, you can change the T60 in the first line to a higher number, such as

T80 or T110. Be sure, however, to change the tempo for all four voices to be the same; otherwise, the notes will play different lengths and will get out of order. Instead of harmony, you'll get cacaphony.

The fundamental process for programming Amiga music using the MP statements is:

1. Define the voice characteristics (tempo, octave, etc.).
2. Write the notes in four variable statements so the Amiga can play the four variable statements simultaneously.
3. Insert the music into the PLAY program.

The following music programs use this structure and a few interesting variations of it.

Programming a Harmonious "Happy Birthday"

Once again, you can use the program in chapter 7 for the following program. The notes from that program are the simple melody for "Happy Birthday" played as the first voice in a four-voice piece. In addition to the harmony of three more voices, a loop added at the program's end provides an opportunity for a sing-along for two choruses of the song.

```
MP ("T120O2","T120O2","T120O1", "T120O1")
V1$=("G8G8A4G4›C4‹B2G8G8A4G4›D4C2‹G8G8›G4E4
       C8C8‹B4A4›F8F8E4C4D4C2",0)
V2$=("F8F8E4E4E4F2F8F8F4F4B4G2E8E8›C4C4‹G8
       G8F4F4›C8C8C4‹G4B4G2",1)
V3$=("B8B8›C4C4E4D2‹B8B8B4B4›F4E2C8C8E4G4
       E8E8C4C4A8A8G4E4F4E2",2)
V4$= ("G8G8G4G4G4G2G8G8G4G4G4›C2‹G8G8G4›C4C8C8‹
        F4F4F8F8G4G4G4›C2",3)

FOR I=1 TO 2
MP (V1$,V2$,V3$,V4$)
NEXT I
RUN
```

Each time you press the Return key when the cursor is on the RUN line, the "Happy Birthday" tune will play through twice. To play the tune more than twice, change the number 2 in line FOR I = 1 TO 2, to however many times you want "Happy Birthday" to play. Although straightforward here, loops in other songs can be a little tricky. If you tried to add a similar loop in the "Silent Night" program, for instance, the song would not sound the same the second time it was played. It would sound different because the second voice would be one octave lower than in the first chorus. Why the difference? Because the settings at the music's end the first time it was played are retained by the Amiga for the next time the music is played. Thus, at the end of the first chorus the Amiga's four voices were set at octaves 03, 02, 02, and 02, respectively, and that's how it starts the next chorus.

Whenever a second pass through a piece of music doesn't sound right, check the settings at the end of the piece and add the necessary statements (e.g., a › to bring the octaves back up) at the end of the notes to return the Amiga's voices to their original settings at the beginning of the tune. You could also add a new tempo setting at the end of each line speeding up or slowing down the next chorus. Children often find this very funny.

Bach to BASIC

Johann Sebastian Bach was one of the world's greatest musicians. He also may have had a temperament similar to today's computer hackers. A hacker is a person who tries to get a computer to do things for which it was not designed but nevertheless does very well. A few of Bach's most famous contributions to music fall easily into the same definition.

Before Bach's time, keyboard instruments were played with only four fingers held flat on the keys. Bach taught his students to curve their fingers and use their thumbs to play as well. His major technical achievement, however, was to retune (called tempering at the time) harpsichords and other keyboard instruments to play notes in a then unconventional manner. Conventional tempering, known as natural tempering, called for an instrument to be tuned to a single key, such as the key of C or B flat or G. Music played in the key of the tuning sounded wonderful, but played in any other key sounded terrible. To play a tune in a key other than the tempered key of the instrument meant having to retune it first. Consequently, all the good musicians of the time carried a set of piano wrenches for retuning the keyboard each time they were to play a piece in a new key. Bach had both the musical genius and technical skill to figure out how to retemper keyboards so the constant tuning changes demanded by the natural tempering were unnecessary.

So striking was the new "out-of-tune" sound that soon all keyboards were being retempered. To promote the new style, named "even" or "equal" tempering, Bach wrote a series of preludes and fugues in two 24-piece books entitled *The Well-Tempered Clavier*. Each of the pieces is composed in a different key and only even-tempered keyboards can play the correct notes in the pieces. Today, pianos and other keyboard instruments are tuned using the same tempering developed by Bach over two-hundred years ago. If Johann Sebastian Bach was a hacker, he was one of the most influential ever, computers notwithstanding. The following Amiga program is the first of the forty-eight pieces of music Bach wrote to "sell" his new style of tempering. It is appropriately titled "The First Prelude" and it is played in the key of C major (see Figures 8-1a and b).

THE FIRST PRELUDE IN C MAJOR
by J. S. Ba(si)ch

```
PLAY ("T150O2L16",1)
PLAY ("CEG›CE‹G›CE‹CEG›CE‹G›CE‹CDA›DF‹A›D
    F‹CDA›DF‹A›DFO1B›DG›DF‹G›DFO1B›DG›D
    F‹G›DF‹CEG›CE‹G›CE‹CEG›CE‹G›CE‹",1)

PLAY ("CEA›EA‹A›EA‹CEA›EA‹A›EA›CDF+A›D‹
    F+A›D‹CDF+A›D‹F+A›DO1B›DG›DG‹G›D
    GO1B›DG›DG‹G›DG",1)

PLAY ("O1B›CEG›C‹EG›CO1B›CEG›C‹EG›CO1A›
    CEG›C‹EG›CO1A›CEG›C‹EG›CO1D
    A›DF+›C‹DF+›CO1DA›DF+›C‹DF+
    ›CO1GB›DGBDGB‹GB›DGBDGB",1)

PLAY ("O1GB-›EG›C+‹EG›C+O1GB-›EG›C+‹E
    G›C+O1DA›DA›D‹DA›DO1DA›DA›D‹DA›
    DO1DA-›DFBDFB‹DA-DFBDFB",1)

PLAY ("O1EG›CG›C‹CG›CO1EG›CG›C‹CG›CO1
    EFA›CF‹A›CFO1EFA›CF‹A›CF‹DFA›C
    F‹A›CF‹DFA›CF‹A›CFO0G›DGB›
    F‹GB›FO0GDGA›F‹GB›F",1)

PLAY ("O1CEG›CE‹G›CEO1CEG›CE‹G›CE‹C
    GB-›CE‹B-›CEO1CGB-›CE‹B-›CEO0F›
```

JOHANN SEBASTIAN BA(SI)CH

ALL NOTES ARE 16ths UNTIL THE FINAL CHORD

OCTAVE	2 2 2 3 3 2 3 3 o/ 2 2 2 3 3 2 3 3 o/ 1 2 2 3 3 2 3 3 o/ 2 2 2 3 3 2 3 3
NOTE	C E G C E G C E /o C D A D F A D F /o B D G D F G D F /o C E G C E G C E

OCTAVE	o/ 2 2 2 3 3 2 3 3 o/ 2 2 2 2 3 2 2 3 o/ 1 2 2 3 3 2 3 3 o/ ———
NOTE	/o C E A E A A E A /o C D F+ A D F+ A D /o B D G D G G D G /o ———

OCTAVE	1 2 2 2 3 2 2 3 o/ 1 2 2 2 3 2 2 3 o/ 1 1 2 2 3 2 2 3 o/ 1 1 2 2 2 2 2 2
NOTE	B C E G C E G C /o A C E G C E G C /o D A D F+ C D F+ C /o G B D G B D G B

OCTAVE	o/ 1 1 2 2 3 2 2 3 o/ 1 1 2 2 3 2 2 3 o/ 1 1 2 2 2 2 2 2 o/ ———
NOTE	/o G B- E G C+ E G C+ /o D A D A D D A D /o D A- D F B D F B /o ———

OCTAVE	1 1 2 2 3 2 2 3 % 1 1 1 2 2 1 2 2 o/ 1 1 1 2 2 1 2 2 o/ Ø 1 1 1 2 1 1 2
NOTE	E G C G C C G C % E F A C F A C F /o D F A C F A C F /o G D G B F G B F

OCTAVE	o/ 1 1 1 2 2 1 2 2 o/ 1 1 1 2 2 1 2 2 o/ Ø 1 1 2 2 1 2 2 o/ ———
NOTE	/o C E G C E G C E /o C G B- C E B- C E /o F F A C E A C E /o ———

OCTAVE	Ø 1 1 2 2 1 2 2 o/ Ø 1 1 2 2 1 2 2 o/ Ø 1 1 1 2 1 1 2 o/ Ø 1 1 2 2 1 2 2 o/
NOTE	F+ C A C E- A C E- /o A- F B C D B C D /o G F G B D G B D /o G E G C E G C E /o

OCTAVE	Ø 1 1 2 2 1 2 2 o/ Ø 1 1 1 2 1 1 2 o/ Ø 1 1 2 2 1 2 2 o/ Ø 1 1 2 2 1 2 2 o/
NOTE	G D G C F G C F /o G D G B F G B F /o G E- A C F+ A C F+ /o G E G C G G C G /o

OCTAVE	Ø 1 1 2 2 1 2 2 o/ Ø 1 1 1 2 1 1 2 o/ Ø 1 1 1 2 1 1 2 o/ ———
NOTE	G D G C F G C F /o G D G B F G B F /o C C G B- E G B- E /o ———

		T8
OCTAVE	Ø 1 1 1 2 2 2 1 2 1 1 1 1 1 1 1 Ø Ø 2 2 3 3 3 2 3 2 2 2 2 2 2 2	C3 G2
NOTE	C C F A C F C A C A F A F D F D C B G B D F D B D B G B D F E D	E2

Figure 8-1

```
    FA›CE‹A›CEO0F›FA›CE‹A›CE",1)

PLAY ("O0F + ›CA›CE – ‹A›CE – O0F + ›CA›CE – ‹
    A›CE – O0A – ›FB›CD‹B›CDO0A – ›FB›CD‹B›CDO0G
    ›FGB›D‹GB›DO0G›FGB›D‹GB›DO0G›EG›CE‹
    G›CEO0G›EG›CE‹G›CE",1)

PLAY ("O0G›DG›CF‹G›CFO0G›DG›CF‹G›CFO0G›D
    GB›F‹GB›FO0G›DGB›F‹GB›FO0G›E – A›CF + ‹
    A›CF + O0G›E – A›CF + ‹A›CF + O0G›
    EG›CG‹G›CGO0G›EG›CG‹G›CG",1)

PLAY ("O0G›DG›CF‹G›CFO0G›DG›CF‹G›CFO0G›
    DGB›F‹GB›FO0G›DGB›F‹GB›FO0C›CGB – ›
    E‹GB – ›EO0C›CGB – ›E‹GB – ›E",1)

PLAY ("O0C›CFA›CFC‹A›C‹AFAFDFD‹CBO2GB›DFD‹B›D‹
    BGBDFED",1)

MP ("T150O2G1","O3C1","T150O2E1","")
RUN
```

Although this program looks formidable, it isn't. Except for the first PLAY statement and the MP statement at the end, the rest of the statements are simply the notes of the music which have nothing to do with the actual programming. The piece, however, does illustrate some other programming features of the PLAY statement.

The first line defines the settings for only a single voice because the prelude is played without harmony (until the last line of the program). The first line defines the music to be played at a tempo of 150 (T150) and start at octave O2. The L16 at the end of the line is a new feature of the PLAY statement. It means that all the notes following the L16 will be played as sixteenth notes until a nonsixteenth note is written. But all the notes in the prelude are sixteenth notes, so you can write the piece using just the note names and not the kind of note. For example, in the "Silent Night" program, the notes are written F8 G4 A4, and so forth — a note name followed by a number to indicate the kind of note. But in the prelude, the notes are written CEGDFA, note names not followed by a number. Without the L16 feature, you would have had to write the notes as C16 E16 G16 D16, etc. To "turn off" the statement that all notes will be sixteenth notes, you just return to writing notes as before, e.g., C4 D8; as soon as the Amiga sees a nonsixteenth note, the L16 statement no longer applies.

The L feature works with other note lengths as well. For instance, to have a series of notes played as whole notes, you write an L1 just before the series; notes played as quarter notes can be preceded by an L4. Any of the numbers that define the kind of a note can be used with this L feature.

Although the L feature is an obvious time saver in Bach's prelude, it serves another purpose, too. Some notes can only be played correctly by the Amiga when they are preceded by an L statement. For instance, a half note played as a triplet of three half notes would have to be preceded by an L3 in order to sound right. (Three half notes by themselves would not be played at the proper beat.) The Amiga's L statements used most often are:

L1	Whole note
L2	Half note
L3	One of a triplet of three half notes (1/3 of a 4-beat measure)
L4	Quarter note
L5	One of a quintuplet (1/5 of a measure)
L6	One of a quarter note triplet
L7	One of a seventh
L8	Eighth note
L16	Sixteenth note
L32	Thirty-second note
L64	Sixty-fourth note

Intervening L statements, such as L9, L10, L18, and L30, can also program the Amiga, but are used infrequently in music and have rather abstruse meanings. The L statement cannot be used with a pause (P) statement. Pauses must be followed with a number such as P4 (pause for a quarter-note duration).

One final word about the L statement. Like the rest of the notations available with PLAY and MP, it can be written anywhere in the flow of the music, not only in the first PLAY statement. For instance, if a piece of music had a long passage in its middle consisting of notes all of the same kind, you could write the appropriate L statement and then just the note names. The Amiga would know that all the notes following the L statement would be the same kind, until a note followed by a number was again written.

Although the L statement isn't used within the music lines of the prelude, another notation is. In the fourth PLAY statement, the first notation is for an octave of O1 instead of for a note name. The octave notation is there because the music calls for a G in octave 3 to be followed by a B in octave 1 (the G is at the end of the previous line, the B follows the O1 at the beginning of the fourth PLAY line). A < symbol to lower a note an octave won't work for the B because it must be lowered *two* octaves and a < only lowers the note one octave. Writing the B's octave in front of it does the job. And, at the same time, illustrates how PLAY statement notations can be added to the flow of the notes.

Another feature of the prelude program is that it does not use the V1$, V2$, V3$, and V4$ format of the previous musical examples. Using the straight PLAY statements for the program's lines works because the prelude is played with only one voice, i.e., no harmony. Those lines could have been longer, up to 255 characters per line, but were written as shorter statements so any typing errors could be found more easily. When writing a long piece, breaking the notes up into a series of shorter statements can help when you need to find typos. Also, if the music doesn't sound right, it's easier to find the offending notes when the lines are both shorter and coordinated in some way with the music. For instance, the lines of the prelude program contain the notes for either six or eight complete measures of the composition.

The last line of the program plays a three-voice chord. Voice 1, which has been playing the rest of the music, is programmed to play a whole note C (C1) at octave 3 (O3). Voices 0 and 2 have to be programmed to play at the same tempo (T150); thus their definitions begin with T150. Voice 0 is programmed to play a whole note G at octave 2 (O2G1), and voice 2, a whole note E also at octave 2 (O2E1). Voice 3 is not used, so its definition is the null value " ".

As it's written, the notes for Bach's prelude are relatively straightforward, although keeping the octaves at the right place is a bit tedious. However, the piece presents another programming challenge. Each eight-note measure is repeated once, which means that a looping program could be written to repeat the measures and effectively cut the number of notes you have to type by almost half. Many pieces of music are amenable to similar programming shortcuts provided by the BASIC language.

Some High-Tech Music: An Original Arrangement for the Theme from *Star Wars*

The following arrangement of the *Star Wars* theme uses only three Amiga voices. To remind you of the original movie, you could create "laser sounds" or "R2D2 beeps" on the fourth voice and intersperse them with the music. The music was composed by John Williams, the director of the Boston Pops Orchestra. This "Star Wars" arrangement is not a series of block chords as are "Happy Birthday" and "Silent Night." This selection features "moving voices." For example, the melody moves quickly in eighth notes (voice 1), while voices 2 and 3 are in quarter notes at several points. Also, below a slow melody, voices 2 and 3 may move more quickly. This adds variety and interest to the arrangement. Including moving voices is to encourage you, especially if you're experienced with music, to use the capability of the Amiga to produce two or more voices simultaneously as independent musical lines. The arrangement uses all of the programming steps described earlier. Remember to insert the music into the PLAY program at the correct place (see Figures 8-2a, b,and c).

THE THEME FROM STAR WARS
Composed by J. Williams
Arranged by A.B. Myers

```
MP (" ","T120O2V9","T120O2V8","T120O1V8")

V1$= ("L12DDDG2›D2L12C‹BA›G2D4L12C‹B
    A›G2D4L12C‹B›C‹A2L12DP12L12D
    G2›D2L12C‹BA›G2D4L12C‹BA›G2D4
    L12C‹B›C‹A2L12DP12L12DE4.L8E›C‹
    BAGL12GABAP12L12EF+4L12DP12
    L12DE4.L8E›C‹BAG",1)

V1.1$= ("O3D8.D16‹A2L12DP12DE4.L8E›C‹B
    AGL12GABAP12L12EF+4L12DP12L12D›L8
    GFE-DC‹B-AG›D2.L12‹DDDG2›D2L12C‹BA›G
    D4L12C‹BA›G2D4L12C‹B›C‹A2L12DP12L12D",1)

V1.2$= ("G2›D2L12C‹BA›G2D4L12C‹BA›G2D4
    L12C‹B›C‹A2›D4G4L12‹GGG›G4‹L12G
    GG›G4‹L12GGGG4",1)
```

STAR WARS–MAIN THEME

PAGE # 1

		Ⓧ	①	②	③	④	⑤	⑥
V1	O	2 / /	2 3	3 2 / 3 /	3 2 / 3 /	3 2 3 2 2 2	2 3	3 2 / 3 /
	N	D D D	G D	C B A G D	C B A G D	C B C A D P D	G D	C B A G D
	L	12 / /	2 2	12 / / 2 4	12 / / 2 4	12 / / 2 12 12 12	2 2	12 12 12 2 4
V2	O	2	1 2	2 / /	2 / /	2 2 2 2	1 2	2 2 2
	N	D D D	B G	E G G	E G G	F D D P D	B G	E G G
	L	12 / /	2 2	4 2 4	4 2 4	4 2 12 12 12	2 2	4 2 4
V3	O	1 / /	1 / / /	2 1 2 1	2 1 2 1	1 1 1 1 1	1 / / /	2 1 2 1
	N	D	G F+ E D	C B D B	C B D B	A F+ D D P D	G F+ E D	C B D B
	L	12 / /	4 / / /	4 4 / /	4 4 / /	4 4 4 12 12 12	4 / / /	4 4 4 4

		⑦	⑧	⑨	⑩	⑪
V1	O	3 2 2 3 3	3 2 3 2 2 2	2 2 3 2 2 2	2 2 2 2 2 2 2 2	2 2 3 2 2 2
	N	C B A G D	C B C A D P D	E E C B A G	G A B A P E F+ D P D	E E C B A G
	L	12 12 12 2 4	12 12 12 2 12 12 12	4. 8 8 8 8 8	12 12 12 12 12 12 4 12 12 12	4. 8 8 8 8 8
V2	O	2 2 2	2 2 2 2	1 1 2 2	1 2 1 2 2	1 1 2 2
	N	E G G	F D D P D	G G E C	B C A D P D	G G E C
	L	4 2 4	4 2 12 12 12	4. 8 4 4	4 4 4 12 12 12	4. 8 4 4
V3	O	2 1 2 1	1 1 1 1 1	1 1 1 1	1 1 1 1 1	1 1 1 1
	N	C B D B	A F+ D D P D	C C G E	D E D D P D	C C G E
	L	4 4 4 4	4 4 4 12 12 12	4. 8 4 4	4 4 4 12 12 12	4. 8 4 4

Figure 8-2a

```
V2$= ("L12DDD‹B2›G2E4G2G4E4G2G4F4D2
    L12DP12L12D‹B2›G2E4G2G4E4G2G4F4D2
    L12DP12L12D‹G4.G8›E4C4‹B4
    ›C4‹A4›L12DP12L12D‹G4.G8›E4C4",2)

V2.1$= ("F8.F16D2L12DP12L12D‹G4.G8›E4
    C4‹B4›C4‹A4›L12DP12L12D›C4‹G4E4C+4
    F+2.L12C‹BAB2›G2E4G2G4E4G2G4F4D2L12
    DP12L12D",2)

V2.2$= ("‹B2›G2E4G2G4E4G2G4F4D2F+4G4L12
    DDDB-4L12E-E-E-G4L12DDD‹B4",2)

V3$= ("L12DDDL4GF+ED›C‹B›D‹B›C‹B›D‹B
```

STAR WARS–MAIN THEME PAGE # 2

		12	13	14	15	
V1	O	3 3 2 2 2	2 2 3 2 2 2	2 2 2 2 2 2 2 2	3 3 3 3 3 2 2 2	
	N	D D A D P D	E E C B A G	G A B A P E F+ D P D	G F E– D C B– A G	
	L	8. 16 2 12 12 12	4. 8 8 8 8 8	12 12 12 12 12 12 4 12 12 12	8 8 8 8 8 8 8 8	
V2	O	2 2 2 2 2	1 1 2 2	1 2 1 2 2	3 2 2 2	
	N	F F D D P D	G G E C	B C A D P D	C G E C+	
	L	8. 16 2 12 12 12	4. 8 4 4	4 4 4 12 12 12	4 4 4 4	
V3	O	1 1 1 1 1	1 1 1 1	1 1 1 1 1	2 2 1 1	
	N	B– B– F+ D P D	C C G E	D E D D P D	E– C G A	
	L	8. 16 2 12 12 12	4. 8 4 4	4 4 4 12 12 12	4 4 4 4	

		16	17	18	19	20	
V1	O	3 2 2 2	2 3	3 2 2 3 3	3 2 2 3 3	3 2 3 2 2 2	
	N	D D D D	G D	C B A G D	C B A G D	C B C A D P D	
	L	2. 12 12 12	2 2	12 12 12 2 4	12 12 12 2 4	12 12 12 2 12 12 12	
V2	O	2 2 1 1	1 2	2 2 2	2 2 2	2 2 2 2	
	N	F+ C B A	B G	E G G	E G G	F D D P D	
	L	2. 12 12 12	2 2	4 2 4	4 2 4	4 2 12 12 12	
V3	O	2 1 1 1 1 1 1 1	1 1 1 1	2 1 2 1	2 1 2 1	1 1 1 1 1	
	N	D D D D D E F F+	G F+ E D	C B D B	C B D B	A F+ D D P D	
	L	4 12 12 12 4 12 12 12	4 4 4 4	4 4 4 4	4 4 4 4	4 4 4 12 12 12	

Figure 8-2b

```
     AF+DL12DP12L12DL4GF+ED›C‹B›D‹B›C‹B›
     D‹BAF+DL12DP12L12DC4.C8
     G4E4D4E4D4L12DP12L12DC4.C8G4E4",3)

V3.1$= ("B-8.B-16F+2L12DP12L12DC4.C8G4E4D4E4D4
     L12DP12L12D›E-4C4‹G4A4›D4‹L12DDDD4
     L12EFF+L4GF+ED›C‹B›D‹B›C‹B›D‹B
     AF+L12DP12L12D",3)

V3.2$= ("L4GF+ED›C‹B›D‹B›C‹B›D‹BAF+2›
     C4‹B4L12BBB›E-4‹L12AAAB4L12BBBG4",3)
MP (" ",V1$,V2$,V3$)
MP (" ",V1.1$,V2.1$,V3.1$)
MP (" ",V1.2$,V2.2$,V3.2$)
RUN
```

STAR WARS – MAIN THEME PAGE # 3

		21	22	23	24	25	26	
V1	O	2 3	3 2 2 3 3	3 2 2 3 3	3 2 3 2 3	3 2 2 2 3 2 2 2	3 2 2 2 2	
	N	G D	C B A G D	C B A G D	C B C A D	G G G G G G G G	G G G G G	
	L	2 2	12 12 12 2 4	12 12 12 2 4	12 12 12 2 4	4 12 12 12 4 12 12 12	4 12 12 12 4	
V2	O	1 2	2 2 2	2 2 2	2 2 2	2 2 2 2 2 2 2 2	2 2 2 2 1	
	N	B G	E G G	E G G	F D F+	G D D D B- E- E- E-	G D D D B	
	L	2 2	4 2 4	4 2 4	4 2 4	4 12 12 12 4 12 12 12	4 12 12 12 4	
V3	O	1 1 1 1	2 1 2 1	2 1 2 1	1 1 2	1 1 1 1 2 1 1 1	1 1 1 1 1	
	N	G F+ E D	C B D B	C B D B	A F+ C	B B B B E- A A A	B B B B G	
	L	4 4 4 4	4 4 4 4	4 4 4 4	4 2 4	4 12 12 12 4 12 12 12	4 12 12 12 4	

V1	O	
	N	
	L	
V2	O	
	N	
	L	
V3	O	
	N	
	L	

Figure 8-2c

The program's first line first tells voice 0 not to play — the " " signifies a null definition for voice 0. The rest of the line tells voice 1 (which is for the melody) to play at a tempo of 120 and begin at octave 2. The V9 sets the volume for voice 1 at level 9, the loudest being 15. Voice 2 is set for the same tempo, to begin at octave 2, and to be played at a slightly lower volume (V8) than the melody. Voice 3 is the same as voice 2 except it begins at octave 1 (O1). The lines containing the Vx.x$ statements are the notes of the music; the last three MP lines instruct the Amiga to play the lines of notes in their proper order. Note that the first item in the MP statements is the " " for voice 0.

This program shows how to continue the music statements in case the number of notes gets too long for any one of the lines. V1.1 and V1.2, for instance, identify continuations of voice 1. The 1.1 just identifies voice 1, continuation 1, and the 1.2 identifies voice 1, continuation 2. To the Amiga, it doesn't matter what you name the continuations — they could be X, Y, and Z. But you'll probably find that it helps to keep the names of the continuations (V1.1 and V1.2) similar to the first line (V1) of notes in the set, so you know which lines belong together.

A Final Word about Programming Music

Music is becoming such a popular application of personal computers that several magazines have sprung up devoted solely to the topic. Furthermore, commercial programs designed to make writing music easier than it is with BASIC programming are gaining widespread use. Generally, the applications programs use screen graphics, such as a music staff, as well as an on-screen list of notes to aid in the writing. One of the benefits of knowing how to program music with the PLAY and SOUND statements is that you will be able to better evaluate these commercial programs' capabilities to see if they offer anything that you couldn't do in BASIC.

CHAPTER

9

Creating Sound Waveforms

If you connect an oscilloscope to a microphone and play a note on a musical instrument, the waveform appearing on the oscilloscope screen represents the sound produced by that particular instrument. The waveform may vary a bit depending on the note, but its overall shape remains roughly the same over the instrument's range. That waveform is known as the *characteristic waveform* for that kind of instrument.

The characteristic waveform of a tuning fork is a simple sine wave. Characteristic waveforms of musical instruments have more complex patterns, illustrating the sound's richness and "tone color." Irritating noises, such as chalk grating on a blackboard, or random noise (white noise) produce waveforms with no discernible pattern.

A few characteristic waveforms from different instruments are shown in Figure 9-1.

The programs in this chapter are for creating waveforms easily. After creating an instrument's waveform, you can assign it to an Amiga voice, and that voice will play its sound as if it were that instrument. The music

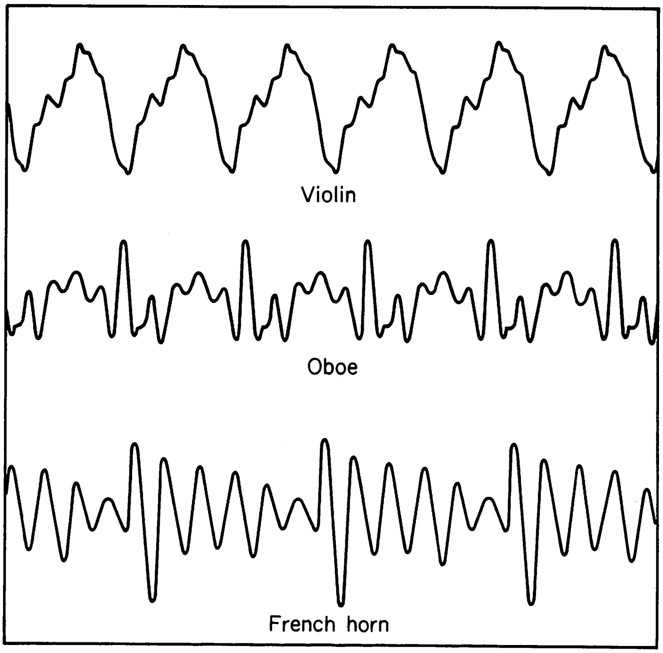


Figure 9-1

you programmed in chapter 8 can then be played as various instruments. Voice 0, for example, could be a banjo, voice 1 a piano, voice 2 a drum, and voice 3 a trumpet. Or perhaps use three voices for instruments and the fourth one for singing. ("Daisy, Daisy" as sung by HAL the computer in the movie *2001: A Space Odyssey*?)

CREATING WAVEFORMS WITH THE WAVE COMMAND

The Amiga BASIC WAVE command creates waveforms. WAVE requires an array of 256 numbers to create the waveform; the numbers plot a curve along the x-axis of a graph to produce a tracing of a wave's shape. Each number of the array represents the height of one point plotted on the curve. The y-axis of the graph is from −128 to 127, which are the lowest and highest numeric values the plotting points can have.

Figuring out all 256 points and typing them in an array is a long job. To shorten the process, the WAVE command has the capability for plotting different sine curves. However, to get the complex plot of an instrument's characteristic waveform, you still need to create the array.

A PROGRAM TO DRAW WAVEFORMS AUTOMATICALLY

The following wave-plotting program, called Wave Shaper, draws a waveform as you drag the mouse across the screen. The program can also produce the array of numbers that represents the drawn curve, and will put the array in a file as a series of DATA statements. You can then use those DATA statements to produce an instrument's characteristic waveform and assign it to one of the Amiga's voices.

Type the program and save it in a file. To use it, load Amiga BASIC, select the file, and run the program. Now drag the mouse and draw a curve from one edge of the screen to the other. Release the button when the curve is as you want it. To modify the curve, merely drag the mouse again to get the curve you want.

Select Play from the menu to hear a brief scale of notes played according to that waveform. Note that before the sound begins, the program calculates all 256 points in the curve; a counter on the screen tells you how the program is progressing. If the notes sound as you want them to, you can assign the waveform to an Amiga voice (as described later).

If you want to draw a new curve to change the sound completely, select Clear from the menu and start again. This program is very sensitive to waveshapes, so getting just the right sound can take a bit of trial and error. But when you've got the one you want, save it in a file to build a "waveform library." Then, as you program music, you can assign waveforms to the voices without having to redraw the curves (see Figure 9-2).

```
WaveShape:
  '
  ' Declare the waveshape array
  '
  DEFINT w
  DIM w(255), s(7)
  '
  ' Define the functions that convert between
screen y-coordinates
```

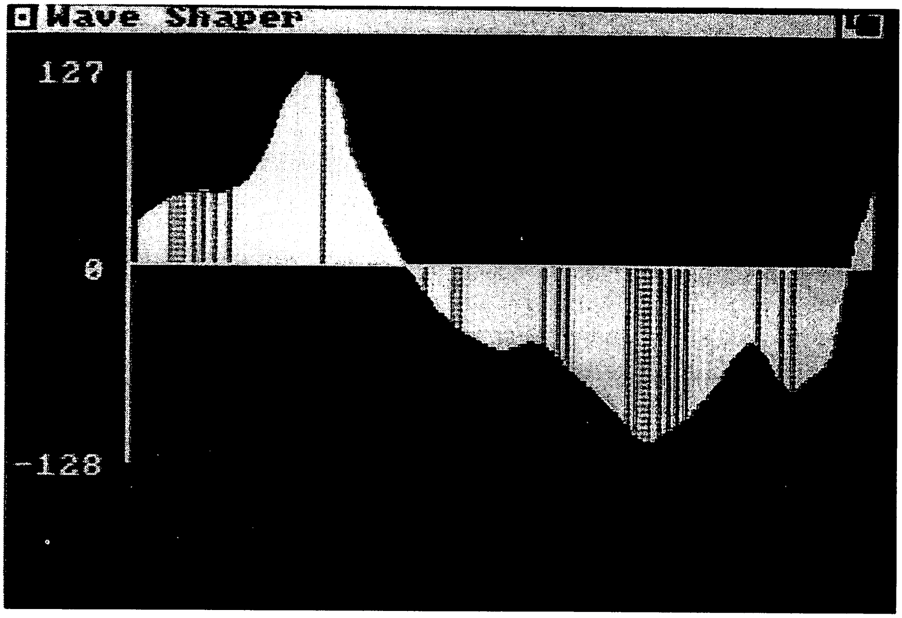


Figure 9-2

```
  ' (0..199) and the waveshape y-coordinates
(127..-128)
  '
  DEF FNSW(Y) = (63 - Y) * 2
  DEF FNWS(Y) = 74 - INT(Y/2)
  '
  ' Set up menu options
  '
  MENU 1,0,1,"Options "
  MENU 1,1,1,"Play"
  MENU 1,2,1,"Write"
  MENU 1,3,1,"Clear"
  MENU 1,4,1,"Quit"
  '
  SCREEN 2,320,200,3,1
  WINDOW 2,"Wave Shaper",,12,2
  '
```

```
   '
   GOSUB ClearWave
   FOR i = 1 TO 7
      READ s(i)
   NEXT i
   DATA 523.25, 587.33, 659.26, 701.0, 783.99,
880.0, 993.0
   '
   ' Finally, set up the interrupt handlers
   '
   ON MENU GOSUB MenuHandler
   ON MOUSE GOSUB MouseHandler
   MENU ON
   MOUSE ON
   '
   ' The infinite loop that follows is the “actual”
program
   ' All the real work is done by the two
“interrupt” handlers
   '
   WaitLoop:
      GOTO WaitLoop
   '
   MouseHandler:
   WHILE MOUSE(0) ‹ 0
      X = MOUSE(1)
      Y = MOUSE(2)
      IF (X‹296) AND (X›39) AND (Y‹139) AND (Y›10)
THEN
      old = w(X – 40)
      IF old ‹› 0 THEN LINE (X,73) – (X,FNWS(old)),0
      LINE (X,73) – (X,Y),1
         w(X – 40) = FNSW(Y – 11)
      END IF
   WEND
   RETURN
   '
   MenuHandler:
      IF MENU (0) ‹› 1 THEN RETURN
      Item = MENU (1)
      IF Item = 4 THEN
         WINDOW OUTPUT 1
         WINDOW CLOSE 2
         STOP
      END IF
```

```
  ON Item GOSUB PlaySong, WriteSong, ClearWave
  RETURN
'
ClearWave:
  CLS
  '
  ' Draw axis
  '
  LINE (39,11) – (39,138),1
  LINE (40,11) – (40,138),1
  LINE (40,73) – (295,73),1
  '
  ' Title it
  '
  LOCATE 2,1: PRINT “ 127”;
  LOCATE 10,1: PRINT “ 0”;
  LOCATE 18,1: PRINT “– 128”;
  '
  ' Zero out the waveshape table
  '
  FOR i = 0 TO 255
    w(i) = 0
  NEXT i
RETURN
'
PlaySong:
  WAVE 1,w
  FOR i = 1 TO 7
    SOUND s(i),10,,1
  NEXT i
RETURN

WriteSong:
  LOCATE 22,1: INPUT “File ”;File$
  IF File$ = “” THEN
    LOCATE 22,1: PRINT “                          ”
    RETURN
  END IF
  OPEN File$ FOR OUTPUT AS #1
  FOR i = 0 TO 15
    PRINT #1,“DATA ”;
    FOR j = 0 TO 15
      PRINT #1,w(i*15+j);
      IF j < 15 THEN PRINT #1,“,”; ELSE PRINT #1,“”
    NEXT j
    NEXT i
```

```
CLOSE #1
LOCATE 22,1: PRINT " ^
RETURN
```

When you're finished drawing waveforms, select the Quit option from the menu.

ASSIGNING WAVEFORMS TO AMIGA VOICES

Although it doesn't print them on the screen, the program can produce a series of DATA statements containing the 256 numeric elements of the array that you would have had to type using the WAVE command. These numeric elements are what the Amiga needs to make one voice sound like a different instrument.

Assigning waveforms to the Amiga's voices takes four steps:

1. Draw the waveform with the Wave Shaper program and save the waveform's 256-element array in a file.
2. Load the PLAY program of the previous chapter.
3. Enter a short looping program in the PLAY program for each voice that you want to assign a waveform.
4. Merge the files containing the DATA statements for that voice into the PLAY program.

Step 1. As described, draw the waveform you want, then select Write from the menu. A prompt asks for a filename. Type a device name and filename such as df1:wavelib1.dat and press the Return key. The DATA statements are written to that file.

Step 2. Select Quit from the Wave Shaper menu and load the PLAY program.

Step 3. In the PLAY program, find the two REM statements at the beginning that read "This is where the programs for assigning waveforms to the different voices will go." At that position, enter the following looping program:

```
FOR i = 0 TO 255
READ W%(i)
NEXT i
WAVE n,W%
```

Enter the voice number that you want to assign a waveform to at the *n* in the line WAVE *n*,W%. For instance, to assign the waveform to voice 3, type the program as WAVE 3,W%. If you want to assign waveforms to more than one voice, enter a separate looping program for each voice. Use a different variable name for each voice. That is, change W% in the second looping program to another name such as T% or V%. Make sure to change it in both the READ and WAVE statements. The values for *n* must be different as well.

If the program doesn't have music in it yet, add the CALL PLAY and CALL MP statements to the program. Those statements must be after the looping programs.

Step 4. The last step to assigning the waveform to a voice is to merge the DATA statements into the PLAY program. To merge one program into another, you use the MERGE command as described in the Amiga BASIC manual. At the end of the looping programs, but before the CALL PLAY and CALL MP statements, type MERGE and the name of a file holding one waveform's DATA statements. The filename is the one you typed after selecting the Write option in Step 1. For example: MERGE df1:wavelib1.dat and press the Return key.

If you are assigning waveforms to multiple voices, add a MERGE statement for each file. Make sure the order of the MERGE statements is the same as the order of the looping programs in Step 3. For instance, if the first looping program assigns a violin waveform to voice 3, the first MERGE statement must name the file holding the DATA statements for the violin waveform.

Once you have assigned all the waveforms to the voices, and entered the music, run the PLAY program. If all four voices have their own waveforms, you'll hear a four-instrument quartet.

A PROGRAM TO DRAW COMBINED WAVES

This program, called Plot Wave, is similar to the Wave Shaper in that it produces waveforms for the Amiga. The main difference is that instead of drawing the waveforms on the screen, you create them as the sum of two mathematical equations. The equations can be trigonometric, such as sin (x) and cos (2x-1), or simply arithmetic. The program then draws the waveform for you.

Type this program into a file and save it. To use the program, load BASIC and load the file, and then run the program. The lines for changing the mathematical equations are the last two in the program. Change the equations and rerun the program.

Waveforms of summed mathematical equations that are pleasing to the eye generally tend to be pleasant to the ear as well. "Rounded tones" often have sweeping, round waveforms, while harsh sounds tend to have jagged, angular waveforms. Try different equations to find pleasing tones.

```
PlotWave:
  SCREEN 2,320,200,3,1
  WINDOW 2,"Summed Wave Plot",,12,2
  CLS
  LOCATE 4,4
  PRINT "Click mouse any time to quit"
  ON MOUSE GOSUB MouseHandler
  MOUSE ON

  DIM s(256), wp%(256), n(8)
  '
  ' Read in note values
  '
  FOR i = 1 TO 8
    READ n(i)
    NEXT i
    DATA 220, 247, 262, 294, 330, 349, 392, 440
    '
    pi = 3.1415926535#
    e = 2.7182818285#
    tp = pi / 128
    x = 0
    '
    ' Compute values for both waves and add them
together
    ' Also find lowest and highest sum values
    '
    FOR i = 0 TO 255
      IF (i MOD 10) = 0 THEN
        LOCATE 6,4
        PRINT i;
      END IF
      GOSUB Waves
      sum = wave1 + wave2
  IF i = 0 THEN
    max = sum
    min = sum
  ELSE
    IF sum > max THEN max = sum
    IF sum < min THEN min = sum
  END IF
  s(i) = sum
  x = x + tp
NEXT i
'
' Compute values for scaling
'
max = CINT(max)
min = CINT(min)
delta = (max - min)/180
d2 = (max - min)/256
hw = (max + min)/2
'
' Indicate highest, middle, and lowest values
'
CLS
LOCATE 1,1: PRINT max;
LOCATE 13,1: PRINT hw;
LOCATE 23,1: PRINT min;
'
' Plot wave on screen
' Also initialize wp%() for note playing
'
FOR i = 0 TO 255
  LINE (40+i,97)-(40+i,97-(s(i)-hw)/delta)
  wp%(i) = CINT((s(i)-hw)/d2)
  IF wp%(i) > 127 THEN wp%(i) = 127
  IF wp%(i) < -128 THEN wp%(i) = -128
NEXT i
'
' Now play A B C D E F G A with the computed wave
'
WAVE 1,wp%
FOR i = 1 TO 8
  SOUND n(i),3,127,1
NEXT i
'
' Now just sit and wait for a mouse click
'
```

```
WaitLoop:
  GOTO WaitLoop
'
' Quit on mouse click
'
MouseHandler:
  WINDOW OUTPUT 1
  WINDOW CLOSE 2
  STOP
'
' The two wave equations are placed here for ease
  of modification
'
Waves:
  wave1 = SIN(x)
  wave2 = SIN(100 * x)
RETURN
```

Some examples of waveforms produced with this wave-summing-program are shown in Figures 9-3a, b, and c.

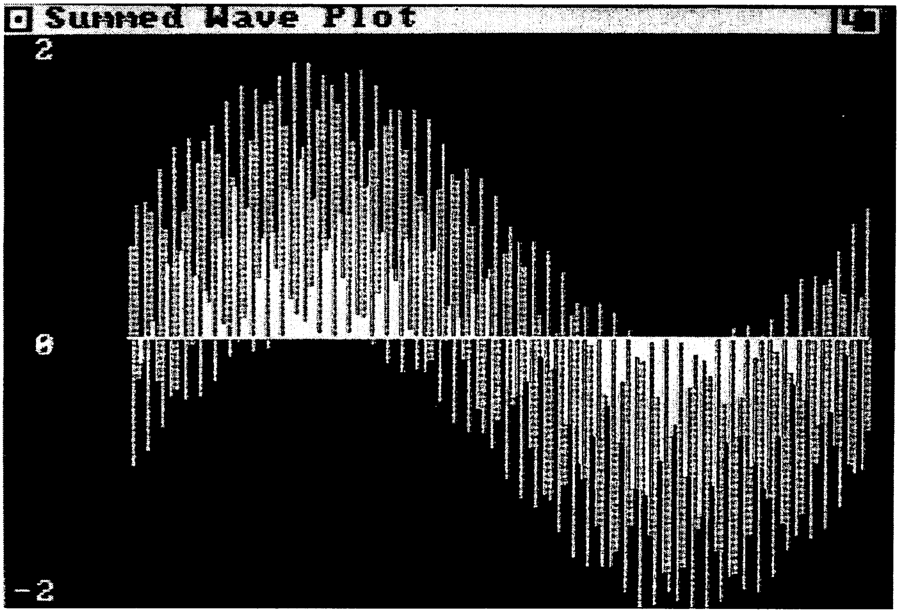


Figure 9-3a

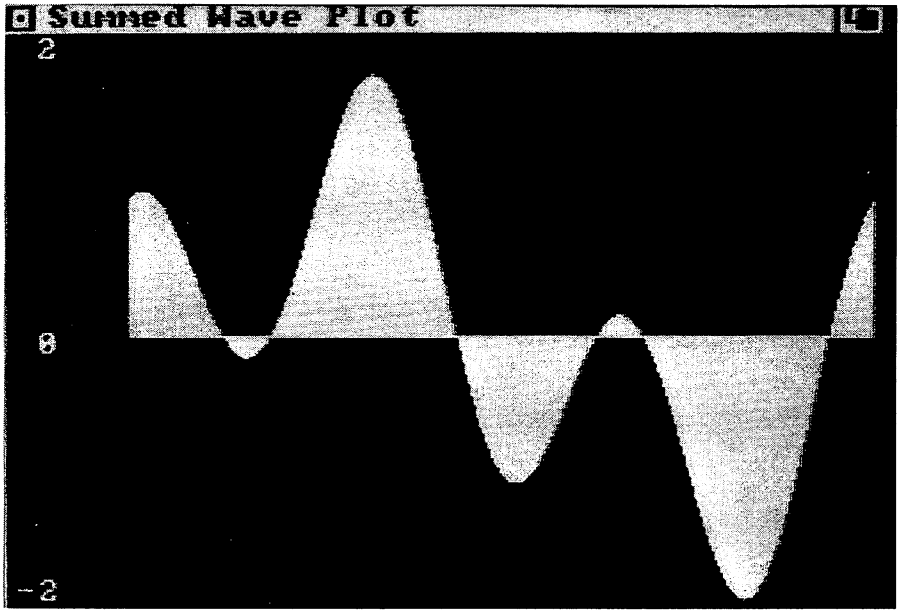


Figure 9-3b

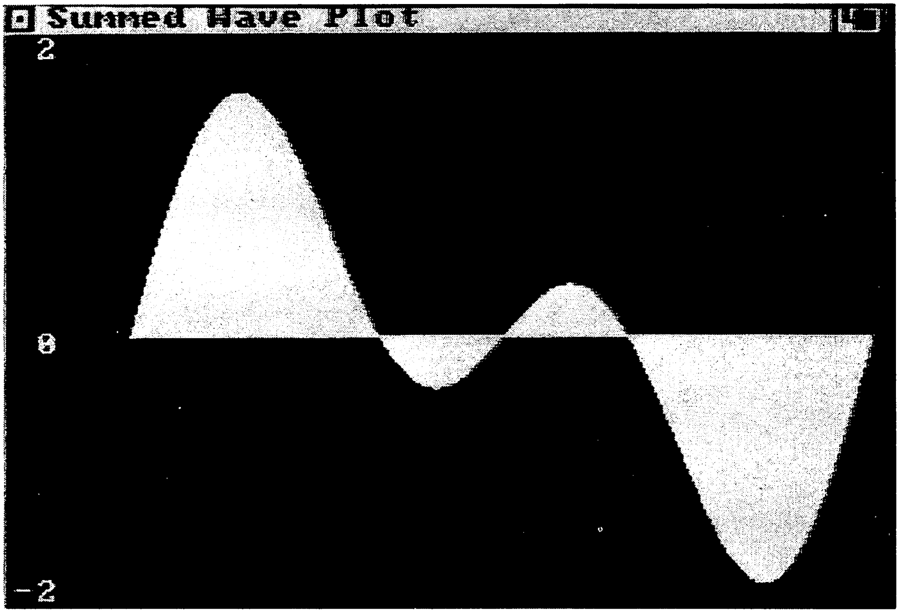


Figure 9-3c

Part IV

TELECOMMUNICATING WITH AMIGA

CHAPTER 10

Reaching Out On-Line

Telecommunications today is big business. Using one of the commercial services, you can see stock market quotes almost as fast as a broker does, news from the AP or UPI newswires, weather reports, sports scores, abstracts of business reports, or any of hundreds of different types of information. These services usually cost an initial sign-up charge, charges for each time you use the service and sometimes a monthly fee. There is also the cost of the telephone call for the connection.

Private networks (known as bulletin boards), on the other hand, are usually free, except for the phone call. Run by dedicated computer owners called system operators, or "sysops" for short, the private networks specialize in any number of topics. For Amiga owners, typical bulletin boards carry dialogues concerning Amiga capabilities, announcements of new programs, or questions about operating different Amiga features. Many of the private bulletin boards also offer free Amiga software.

Although computer telecommunications is a relatively recent phenomena, the underlying concepts are over a hundred years old. It all began in the 1800s with the telegraph.

HOW COMPUTER COMMUNICATIONS CAME TO BE

In 1837 Samuel Morse (of Morse code fame) invented the first practical telegraph. It converted long or short taps (the dots and dashes of the Morse code) on a key to electrical pulses that were sent over wires to a receiving key. As the electrical pulses passed through the receiving key, the same pattern of long and short taps was duplicated enabling a telegraph operator to decipher the message. Then in 1882 a Frenchman, Emil Baudot, borrowed the concept of the telegraph and invented the Teletype, considered the true forerunner of today's data communications. Instead of a key that tapped, he devised and built a rotating distributor much like the distributors in today's automobiles.

Baudot's distributor rotated ten times per second and could send out five pulses of electricity of equal length each second. He developed a code by which letters and numbers could be represented by the five pulses, but rather than having the code consist of long or short pulses, it was based on whether an electrical current was turned on or off during the time when any of the five pulses could be sent. It worked like this: one pulse of electricity (called the start bit) was sent to the receiving distributor to start it rotating. (An on/off pulse is nothing more than the familiar bit of a present-day computer byte.) Then, during each of the five equal time segments when pulses could be sent, a current in the sending distributor would be either on or off in a code pattern predefined to represent a keyboard character. Since the receiving distributor could determine the same five-time intervals and whether the current was on or off during each one, the receiving distributor could duplicate the pattern of on and off currents and thus the code. The sending receiver then stopped for a specific length of time (known as the stop element) to allow the receiving distributor to get ready for the next start bit. After the stop element, the next start bit was sent, followed by the five-pulse code for the next character. The process would continue until all the characters of a message were sent and received.

As they were received, the electrical pulse codes caused a character to be printed on a paper tape, the familiar ticker tape (as in ticker tape parades in New York City). Although still in use in some very old Teletype machines, the Baudot code has been largely replaced by the ASCII code. Computer communications concepts, however, are almost exactly the same. In fact, the rate at which computers send data to one another is called the *baud* rate in honor of Emil Baudot.

Baudot's communications method is known as *asynchronous* because the distributors did not necessarily have to be synchronized to some external clock. As long as the start bit and stop element were recognized, time delays between the sender and receiver were of no consequence. More modern methods involve synchronous communications in which data is sent based on the occurrence of ticks on the computers' internal clocks. Data is transmitted more rapidly with a synchronous connection, but the expense of such communications is currently high. Today, almost all communication between personal computers is asynchronous following the principles established by Baudot.

When computers began to proliferate in the 1950s, and the necessity of linking them together became apparent, engineers developed a method using Baudot's concepts to send character codes over the telephone lines. The problem was that telephone lines are designed to transmit electrical signals which represent sounds, not simply on/off pulses of electricity. The solution to the problem is the *modem* which translates on/off pulses into high- and low-frequency tones that lie within the range of sounds the telephone system can transmit. The sending modem modulates the tones and the receiving modem demodulates them. In fact the word modem is a contraction of modulate-demodulate. Directing how the modem works with the computer is the job of communications software.

Communications software establishes a series of rules which both the sending and receiving computers must follow if data is to be correctly transmitted. The rules, known as the communications protocol, consist of such things as the baud rate of the transmission between the computers, how to handle overload conditions, and how to check for errors.

PUTTING THE AMIGA ON-LINE

To connect an Amiga to a bulletin board or a commercial information service requires a modem, telephone, and communications software.

The Modem

Modems come in three standard forms: internal, acoustic couplers, and direct-connect externals. An internal modem is a circuit board containing all the electronic components necessary for modulating and demodulating

A modem connects a computer to the telephone lines, but that's not the only method of computer communications. You can also communicate with other computers on networks known as local area nets, or LANs for short. Often LANs connect a number of small computers to a large mainframe through "hardwiring," meaning that they directly connect to each other by a wire. Large businesses and the government use a lot of LANs. Some LANs are also national and use the telephone lines too, but in a different manner than modem communications. The idea behind them, though, is the same. Your Amiga could be a "node" (participant) on a LAN, but the necessary hardware and software to make all the connections cost more than the Amiga itself.

a telephone's tones. The board also holds a plug outlet for connecting a telephone cable directly to it. On other computers an internal modem is mounted inside in a dedicated slot on the computer's main circuit board. Because the Amiga has the connector for direct access to the CPU board, an internal modem attaches there as if it were inside the computer itself.

The main advantage of an internal modem is that it doesn't commandeer the serial port (called the RS-232), which is used for connecting some printers and other peripherals. Having a printer and modem connected to a single serial port is possible but requires a switchbox so you can switch between operating one or the other piece of equipment. Normally you won't be able to run them simultaneously, e.g., to print a message as it is being received over the telephone lines.

An acoustic coupler modem looks like a small box with two soft rubber cups, called a cradle, on its top. Tones sent from the Amiga activate a tiny speaker in the modem's cradle cup that holds the telephone mouthpiece; received tones pass through the cradle cup for the earpiece. Acoustic couplers attach to the Amiga's serial port.

These modems have formed the bulwark of models sold for home computers. Their low price, availability, and general ease of operation have long made them favorites. Also, for a number of years direct-connect modems for personal computers were not allowed on the telephone lines by the telephone company.

Although in widespread use, acoustic couplers do have their drawbacks. They need a separate power supply and electrical wall outlet. Also, they fit only standard telephone handsets. For instance, decorator telephones or those with oddly shaped earpieces and mouthpieces do not fit into the cradles. Another problem is their susceptibility to extraneous sounds while in operation. The transmitted extra sounds can disrupt the normal transmission of data. Few new acoustic couplers are being introduced on the market. Their former predominance is now the purview of direct-connect external modems.

Direct-connect external modems look somewhat like the acoustic couplers except they have no cradle for connecting the telephone handset. A cable from the modem attaches directly to a telephone wall jack; another cable attaches to the Amiga's serial port. The data passes directly over these cables. Some direct-connect external modems have DIP (dual in-line package) switches that you have to set before connecting the modem to the Amiga. Occasionally, you also may have to reset the DIP switches when communicating with different types of computers or changing the communications protocol. Resetting the DIP switches requires a screwdriver and can take a fair amount of patience.

Direct-connect external modems are more reliable than the acoustic couplers because no extraneous noises interfere with the tones. These modems also tend to have more features, including the ability to transmit data at higher speeds.

Modem Features

The main features to look for in a modem are baud rate, full/half duplex, automatic answer, and automatic originate. Also important are how it connects to the Amiga and which telephone standard the modem uses.

Baud rate. Baud rate is the speed at which data can be sent through the modem and over the telephone lines. Just as with Baudot's early Teletype, not all of the signals sent through a modem are strictly for data. In today's modems, a start bit is transmitted to signify that data is to follow, then a 7-bit ASCII code representing a unique character is sent, followed by a parity bit to check that the code was correctly received. A stop bit then signifies that the communication is to stop until the next start bit is sent. Start bit, parity bit, 7-bit code, and stop bit add up to 10 bits that have to be sent for each character that is transmitted. The rate at which the 10 bits are sent and received is known as the baud rate.

Technically, baud rate is what engineers call "the number of electrical events" that occur when data is being transmitted. This includes all the bits, but also includes other electrical on/off pulses and timing intervals that modems use when communicating. Practically speaking, baud rate is synonymous with bits per second (bps) as long as the rate is under 1,200 baud. Although most people use baud rate and bps interchangeably — i.e., 1 baud is 1 bit per second — the terms are not quite synonymous.

The Amiga can handle a maximum baud rate of 19,200; however, rates above 4,800 are generally for synchronous communications. The higher-speed modems usually cost more and may require high-quality telephone transmission lines, since any slight interruption in the transmission of data will cause loss of lots of bits. Practically speaking, an Amiga modem's maximum rate for most personal uses is 2,400. The three most common baud rates available with modems for microcomputers are 300, 1200, and (recently) 2400. Many modems can transmit at different baud rates and have a switch for selecting among them.

To communicate, modems must be set to the same baud rate. If a modem is set at 300 baud but receives data sent at 1,200, no proper communication occurs. Having a modem with the capability of multiple baud rates gives the Amiga the capability to communicate with a wider range of other computers. In everyday terms, baud rate by itself has little meaning. It needs to be related to the data that you want to send or receive. For instance, suppose you have a modem that transmits at 300 baud; since 10 bits represent 1 character and 300 baud is about equivalent to 300 bits per second, the modem can transmit 30 characters per second. A standard 8 1/2-by-11-inch page of double-spaced typewritten paper contains about 1,500 characters; thus that one page of text would take about 50 seconds to transmit. A modem transmitting at 1200 baud would take about 13 seconds to do the same job.

A trade-off calculation when purchasing a modem is how much more telephone calls will cost you for the extra time spent transmitting with the lower-speed, lower-cost modem. If you plan to have the computer telecommunicating often, the initial higher cost of the higher-speed modem may quickly be offset by lower telephone bills. By calculating the time it takes to transmit data and multiplying that by the cost of a telephone call, you'll get some rough idea of how much it costs to telecommunicate.

Full/half duplex. Everyday telephone conversations are simultaneous two-way communications; both parties can talk and hear at the same time. This kind of communication is known as *full duplex.* Citizen's band radios and some speaker telephones do not have this capability; you can either talk or listen but not both simultaneously. On CB radios you signify that you're done talking by saying "over" (or in today's trucker slang "c'mon"), letting the other person know that it's his or her turn to talk. This "one way at a time" communication is called *half duplex.* Another kind of one-way communication is called *simplex.* This is how your radio or TV works — it can receive but not transmit.

Modems are normally half or full duplex. Simplex modems are usually for Teletypes and other electronic equipment. Full-duplex modems are generally more expensive but have a feature, known as echoing, that helps maintain accurate communications. Echoing is a way to check that the data is being sent and received correctly. For instance, when data is sent from one computer to another, the sending computer cannot be totally sure the data was received correctly. Needed is a method to compare what was sent with what was received. With full-duplex communications, computer A transmits data to computer B, and then computer B retransmits the same data back to computer A for comparison. Computer B is said to be echoing the data that it received. Echoing is like having a conversation with another person who listens to what you have to say and then says, "To make sure I've got this right, let me repeat what you've just said." Full-duplex modems can echo; half-duplex modems cannot.

Turning the echoing on (many modems have the capability to turn echoing on and off), from your point of view, makes the modem behave as if it were a half-duplex device. That is, by echoing the data that was initially transmitted, the computers are carrying on a full-duplex conversation, but you must wait until that conversation is done before transmitting or receiving further data. In essence, the computers tell you "over" and then you can type in more data to be transmitted. It comes down to a half-duplex conversation as far as you're concerned.

Automatic answer. This feature of a modem, usually known as auto answer, makes your Amiga act like a telephone answering machine. You turn on the computer, prepare it for receiving data sent over the telephone lines, switch on the auto answer button on the modem, and then leave it alone. Messages sent to the computer are automatically received and stored in the Amiga's memory. You do not have to attend the machine. Upon returning to the computer, you'll be able to tell if any data has been received, usually by a message on the screen stating that data was sent to your computer and stored in its memory. Auto answer can be switched off so you can operate the modem manually.

The auto-answer feature of telecommunications is a prime example of the value of multitasking. Your Amiga can receive messages while you're working on something else, giving you the option of manually taking the call or electing to have it routed to your machine. The communications software, running at the same time as your other programs, will not interrupt your work.

Automatic originate. Like auto answer, this feature of a modem is for its unattended operation. You can prepare data to be sent but have the computer send it when the telephone rates are lower. Auto originate will dial the proper telephone numbers automatically, make the connections with the other computer, and transmit the data by itself. Auto originate can be switched on or off.

Telephone standards. Modems are sometimes grouped by their telephone standards. Known as Bell System standards, they define the tone frequencies and signal characteristics the modem must produce to be compatible with the telephone system. The most common standards are Bell 103 and 212A for, respectively, 300 and 1,200 baud, full-duplex modems. Other standards are Bell 113 A and B and 202 for modems that operate at 300 and 1,200 baud, half duplex. The modem you acquire for the Amiga must meet one of the standards in order to work on the Bell system.

The Telephone

Often overlooked when planning to connect an Amiga to the telephone lines, the telephone itself needs to be considered as part of the system. Some modems work with touch tone telephones, but not with rotary dial. Most modems, however, have provisions to work with both types. The jargon phrase for tone dialing is DTMF which stands for "dual-tone modulated frequency"; rotary telephones are indicated by the phrase "pulse dialing."

In businesses, telephone service routed through a switchboard or extension number may not be sufficient for computer communications. Computer-to-computer communication usually requires a direct connection. For example, auto originate and auto answer won't work with switch-boards because the computers are expecting a conversation in electrical signals and don't know what to do when a person answers.

The quality of the telephone line can also be important. A poor-quality connection may ruin telecommunications, especially if static or other noises constantly occur on the line. A few years ago the telephone company suggested sending electronic data only over expensive, high-quality lines. With the proliferation of home computers and their linkups with other computers, this suggestion has been quietly dropped.

Some of the newer features of telephones can also upset computer communications. Call Waiting, for instance, interrupts calls with a clicking noise to signify another incoming call. Computers try to interpret the clicks as data, causing a loss of the real data. The auto-dialing feature of a telephone (it automatically dials and redials a number until someone answers) can also ruin computer communications.

Communications Software

If all computers were the same, there would be little need for the many types of communications software available today. All of the instructions for transmitting data would be in each computer's ROM. But because of the differences among computers, more communications software programs are currently available than brands of computer.

The first and most crucial trait of the software is the choices of settings it provides for the Amiga and its modem. As described earlier, three main modem features are baud rate, full or half duplex, and echoing. Other features are parity checking, number of data bits per character, and a feature known as *flow control*.

Parity checking is a method of verifying that data is transmitted correctly. The sending computer adds an extra bit to each byte so the total number of bits per byte is always either odd or even. The receiving computer checks for the odd or even condition — the byte's parity. If a bit was lost, the parity is incorrect and the computer knows a transmission error occurred. Communications software usually offers you the choice of parity set at odd, even, or none.

The number of data bits per character describes the number of bits used to describe each character. Normally the choice is 7 or 8. The Amiga usually will use 7. In computer jargon this is sometimes called the "word length" (a character being a complete word to a computer); at other times the length is merely referred to as "data bits."

Flow control assures that the amount of data sent to a computer does not overwhelm its capabilities. When data is transmitted, the receiving computer temporarily stores the data in a RAM buffer. A buffer is simply a memory area for holding the received data until the computer can process it. For slow transmission rates, the receiving computer can process data as it arrives, the buffer never fills, and data can be transmitted continuously. But if the transmission rate is high (usually 1,200 bps or over), data is received faster than it can be processed, filling the buffer. The receiving computer then has to tell the sending computer to stop transmitting until the data in the buffer can be processed. Flow control is the method computers use to stop transmitting when the buffer is full and to start again once the buffer is empty.

The traditional commands of flow control are XON and XOFF. XON stands for "transmission on" and XOFF means "transmission off." XOFF is usually ASCII code for Ctrl-S; XON is usually ASCII code for Ctrl-Q. With the flow control option turned on, the computers send the XON/ XOFF messages automatically. You do not have to type any of the keys manually.

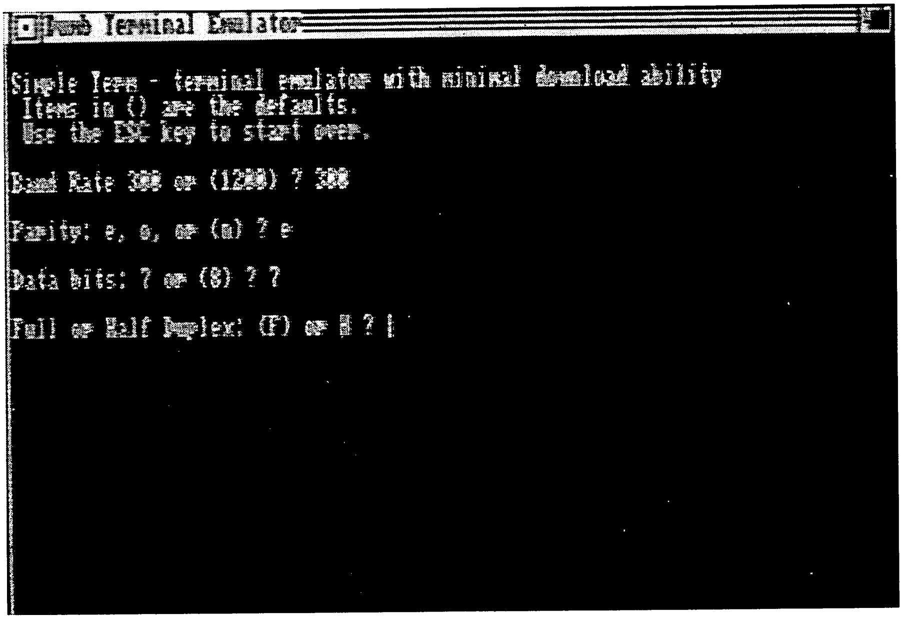
Other typical options of communications software are for setting the screen to 40 or 80 characters, and for using AmigaDOS to find and retrieve files during the communication connection.

A FREE COMMUNICATIONS SOFTWARE PROGRAM

Many Amiga bulletin boards offer a free communications software program known as *AmigaTerm*. It works with most modems and has many of the features you will need for telecommunicating. There are two items you'll need to resolve before using *AmigaTerm*: (1) you'll need some communications software to connect to the bulletin boards in order to get *AmigaTerm* downloaded to your machine and (2) *AmigaTerm* needs to be compiled with the *Lattice C* compiler program.

The following two programs can be used as communications software for your Amiga. The first one makes the Amiga a "dumb terminal" and is for connecting with a bulletin board. You'll be able to send and receive messages, but you can't download files. The second program is a "terminal emulation" program that provides the ability to upload files to the network and download files to your Amiga. To run these programs, you only need a modem already connected to your Amiga. After getting the *AmigaTerm* program in one of the downloaded files, get it compiled by either purchasing the compiler program yourself or finding a friend (or bulletin board acquaintance) who will compile the *AmigaTerm* program for you.

THE DUMB TERMINAL PROGRAM



```
Simple Term - terminal emulator with minimal download ability
 Items in () are the defaults.
 Use the ESC key to start over.

Baud Rate 300 or (1200) ? 300

Parity: e, o, or (n) ? e

Data bits: 7 or (8) ? 7

Full or Half Duplex: (F) or H ? 
```

Figure 10-1

```
InitDumbTerm:
   DEFINT a-z
   SCREEN 2,640,200,1,2
   WINDOW 2,"Dumb Terminal Emulator",,31,2
   '
```

```
  ' Set up constants and flags
  '
  true = (1 = 1)
  false = NOT true
  ESC$ = CHR$(27)
  '
  ' Let user select communications parameters
  '
  CLS
  PRINT
  PRINT “Simple Term - terminal emulator with
            minimal download ability”
  PRINT “ Items in (  ) are the defaults.”
  PRINT “ Use the ESC key to start over.”
  PRINT

GetBaudRate:
  INPUT “Baud Rate 300 or (1200) ”;baud$
  IF baud$ = ESC$ THEN InitSimTerm
  IF baud$ = “” THEN baud$ = “1200”
  IF (baud$ ‹› “300”) AND (baud$ ‹› “1200”) THEN
    PRINT “** ”;
    GOTO GetBaudRate
  END IF
  PRINT

GetParity:
  INPUT “Parity: e, o, or (n) ”;parity$
  IF parity$ = ESC$ THEN InitSimTerm
  IF parity$ = “” THEN parity$ = “n”
  IF (LEN(parity$) › 1) OR (INSTR(“eonEON”,parity$)
     = 0) THEN
    PRINT “** ”;
    GOTO GetParity
  END IF
  PRINT

GetDataBits:
  INPUT “Data bits: 7 or (8) ”;dbit$
  IF dbit$ = ESC$ THEN InitSimTerm
  IF dbit$ = “” THEN dbit$ = “8”
  F (dbit$ ‹› “7”) AND (dbit$ ‹› “8”) THEN
    PRINT “** ”;
    GOTO GetDataBits
  END IF
  PRINT

GetDuplex:
  INPUT “Full or Half Duplex: (F) or H ”;duplex$
  IF duplex$ = ESC$ THEN InitSimTerm
  IF duplex$ = “” THEN duplex$ = “F”
  IF (LEN(duplex$) › 1) OR (INSTR(“fhFH”,duplex$) =
    0) THEN
    PRINT “** ”;
    GOTO GetDuplex
  ELSE
    HalfDuplex = (INSTR(“hH”,duplex$) › 0)
  END IF

'
' Now that the selections have been made, open the
serial port
'
  OPEN “com1:“ + baud$ + ”,“ + parity$ + ”,“ + dbit$ + ”,1” AS
1
  CLS
  PRINT “com1:“ + baud$ + ”,“ + parity$ + ”,“ + dbit$ + ”,1”

TerminalLoop:
    WHILE LOC(1)‹›0
      i$ = INPUT$(1,1)
      PRINT i$;
  WEND
  i$ = INKEY$
  IF i$ = ESC$ THEN EndTerminal
  IF i$ ‹› “” THEN PRINT #1,i$;
  IF HalfDuplex THEN PRINT i$;
GOTO TerminalLoop

EndTerminal:
  WINDOW OUTPUT 1
  WINDOW CLOSE 2
  STOP
```

As described by the remark statements throughout the program, different sections create settings for baud rate, parity, data bits, and duplex. The actual communications section of the program is labeled “Terminal Loop.” It simply waits for a character to be received, and when one is, it's printed to the screen. If the character is the Esc key, the program stops.

To use this program on your Amiga, type it into a BASIC file. When you're ready to communicate with a bulletin board:

1. Make sure the modem is connected and turned on.
2. Load the program from its file (double-click on its icon).
3. On the Output screen, type Run; the choices for the communications parameters (baud rate, parity) appear one at a time.
4. Fill in the choices for each of the parameters; to select the default parameters, press Return (for many bulletin boards, the default values are the ones you want).
5. Press the CapsLock key (many modems only accept uppercase characters (Hayes modems and Hayes-compatible modems have this restriction).
6. Type ATDT if you have a Hayes or Hayes-compatible modem (see your modem instructions for the characters you type for other brands of modems). The ATDT is for automatic dialing by the modem. If you have an acoustic coupler, you have to dial the number yourself.
7. Type the telephone number of the bulletin board (or if you have an acoustic coupler, modem dial it on the telephone) and press Return.
8. A message tells you when the dialing is complete. The modem makes various ringing and connection noises.
9. When a message appears saying Connect, press Return. Depending on the bulletin board, a log-in message of some type appears, and you type the log-in sequence of characters and press Return.
10. You then usually type in a name, and sometimes a password. And you're connected.
11. Breaking the connection depends on the bulletin board. Some require that you type *Bye* and press Return; others want you to type *Off* and press Return. The instructions you get when you first join will explain the log-in and log-out procedures.
12. Once the connection is broken and you're back at the Amiga screen, press Esc to end the DumbTerm program. You can now progress with any other Amiga program.

THE SIMPLE TERMINAL EMULATOR PROGRAM

This type of program is also called a simple "ASCII capture" program. It can send and receive ASCII characters, to or from files. Within the program, the word *logging* refers to the Amiga receiving characters from the serial port — i.e., the characters are being logged into a file. Then the characters are written from the file to the screen. Use this program to download a file holding the *AmigaTerm* program (Figure 10-2).

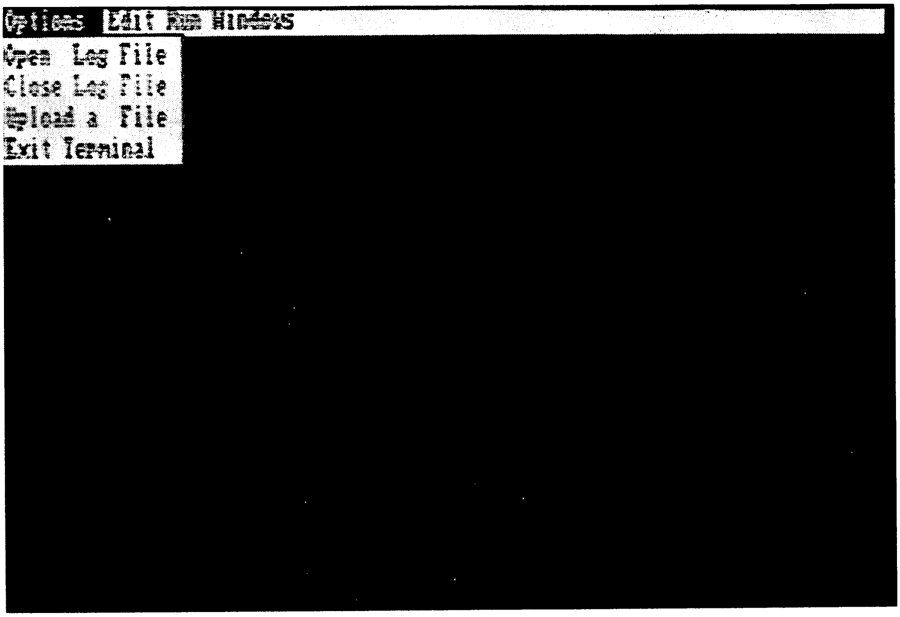


Figure 10-2

```
InitSimTerm:
  DEFINT a-z
  SCREEN 2,640,200,1,2
  WINDOW 2,"Simple Terminal Emulator",,31,2
  '
  ‘ Set up constants and flags
  ‘
  true = (1 = 1)
  false = NOT true
  ESC$ = CHR$(27)
  logging = false
  ‘
  ‘ Set up menu and menu interrupt handler
  ‘
  MENU 1,0,1,“Options ”
  MENU 1,1,1,“Open Log File”
  MENU 1,2,0,“Close Log File”
  MENU 1,3,1,“Upload a File”
  MENU 1,4,1,“Exit Terminal”
  ON MENU GOSUB MenuHandler
  MENU ON
  ‘
  ‘ Let user select communications parameters
  ‘
  CLS
  PRINT
  PRINT “Simple Term - terminal emulator with
minimal download ability”
  PRINT “ Items in ( ) are the defaults.”
  PRINT “ Use the ESC key to start over.”
  PRINT

GetBaudRate:
  INPUT “Baud Rate 300 or (1200) ”;baud$
  IF baud$ = ESC$ THEN InitSimTerm
  IF baud$ = “” THEN baud$ = “1200”
  IF (baud$ ‹› “300”) AND (baud$ ‹› “1200”) THEN
    PRINT “** ”;
    GOTO GetBaudRate
  END IF
  PRINT

GetParity:
  INPUT “Parity: e, o, or (n) ”;parity$
  IF parity$ = ESC$ THEN InitSimTerm
  IF parity$ = “” THEN parity$ = “n”
  IF (LEN(parity$) › 1) OR (INSTR(“eonEON”,parity$)
= 0) THEN
    PRINT “** ”;
    GOTO GetParity
  END IF
  PRINT

GetDataBits:
  INPUT “Data bits: 7 or (8) ”;dbit$
  IF dbit$ = ESC$ THEN InitSimTerm
  IF dbit$ = “” THEN dbit$ = “8”
  IF (dbit$ ‹› “7”) AND (dbit$ ‹› “8”) THEN
    PRINT “** ”;
    GOTO GetDataBits
  END IF
  PRINT

GetDuplex:
  INPUT “Full or Half Duplex: (F) or H ”;duplex$
  IF duplex$ = ESC$ THEN InitSimTerm
  IF duplex$ = “” THEN duplex$ = “F”
  IF (LEN(duplex$) › 1) OR (INSTR(“fhFH”,duplex$) =
0) THEN
    PRINT “** ”;
    GOTO GetDuplex
  ELSE
    HalfDuplex = (INSTR(“hH”,duplex$) › 0)
  END IF

'
' Now that the selections have been made, open the
serial port
'
  OPEN “com1:“ + baud$ + ”,“ + parity$ + ”,“ + dbit$ + ”,1” AS
1
  CLS
  PRINT “com1:“ + baud$ + ”,“ + parity$ + ”,“ + dbit$ + ”,1”
  PRINT

TerminalLoop:
  WHILE LOC(1)‹›0
      i$ = INPUT$(1,1)
      PRINT i$;
      IF logging THEN PRINT #2,i$;
  WEND
  i$ = INKEY$
  IF i$ ‹› “” THEN PRINT #1,i$;
   IF HalfDuplex THEN PRINT i$;
  GOTO TerminalLoop
```

```
MenuHandler:
  IF MENU(0) <> 1 THEN RETURN
  ON MENU(1) GOSUB OpenFile, CloseFile, UploadFile,
EndTerminal
  RETURN

OpenFile:
  PRINT
  PRINT “Name of file (Return to cancel) ”;
  INPUT Filename$
  IF Filename$ = “” THEN RETURN
  ‘
  ‘ Turn off Open option and turn on Close option
  ‘
  MENU OFF
  MENU RESET
  MENU 1,0,1,“Options ”
  MENU 1,1,0,“Open Log File”
  MENU 1,2,1,“Close Log File”
  MENU 1,3,1,“Upload a File”
  MENU 1,4,1,“Exit Terminal”
  MENU ON
  OPEN Filename$ FOR OUTPUT AS #2
  logging = true
  RETURN

CloseFile:
  ‘
  ‘ Turn on Open option and turn off Close option
  ‘
  MENU OFF
  MENU RESET
  MENU 1,0,1,“Options ”
  MENU 1,1,1,“Open Log File”
  MENU 1,2,0,“Close Log File”
  MENU 1,3,1,“Upload a File”
  MENU 1,4,1,“Exit Terminal”
  MENU ON
  CLOSE #2
  logging = false
  RETURN

UploadFile:
  PRINT
  PRINT “File to upload (Return to cancel) ”;
  INPUT upload$
  IF upload$ = “” THEN RETURN
  OPEN upload$ FOR INPUT AS #3
  WHILE NOT EOF(3)
  WHILE LOC(1)<>0
      i$ = INPUT$(1,1)
      PRINT i$;
      IF logging THEN PRINT #2,i$;
    WEND
    i$ = INPUT$(1,#3)
    PRINT #1,i$;
    IF HalfDuplex THEN PRINT i$;
  WEND
  CLOSE #3
  RETURN

EndTerminal:
  IF logging THEN CLOSE #2
  WINDOW OUTPUT 1
  WINDOW CLOSE 2
  STOP
```

Unlike the previous program, this one uses the Amiga's menu capabilities. The menus are: Options for setting the communications parameters, Open and Close a log file, Upload a file for sending it to another computer, and Exiting the program. The MENU statements each have three numbers following them. These numbers describe the state of the menus. For example, in the statement MENU 1,2,0, the 1 refers to the first menu on the screen (there is only one for this program), the 2 refers to the selection within the menu (Close log file for this program), and the 0 refers to the state of the menu item (whether it's active or not; a 0 means the item is not active and you cannot select it, a 1 means it is active). Close log file will be inactive, for example, if you haven't opened the file yet (so choosing Close log file makes no sense). The MENU statements in the program are for changing the state of the menu items.

The numbers, #1, #2, and #3, in the program refer to the serial port and files being opened and closed. This program reads characters from the serial port and sends them to a file, or reads from a file and sends the characters to the serial port.

If the Uploading seems to send files with no carriage returns, but with line feeds instead (the text will be arranged haphazardly on the screen), you can add an extra statement to the program to solve the problem. In the Upload section of the program, type in the line:

```
IF i$ = CHR$(10) THEN PRINT#1,CHR$(13)
```

right after the line

```
i$ = INPUT$(1,#3)
```

The process for using this program on your Amiga is similar to the process for using the DumbTerm program, except you have menus and more options for sending and receiving data.

First, type the program into a BASIC file and save it. When you're ready to communicate with the program:

1. Make sure the modem is connected and turned on.
2. Load the program from its file.
3. On the Output screen, type Run; the menu appears.
4. Select the Choices menu and enter the communications parameters (baud rate, parity) that you want.
5. Fill in the choices for each of the parameters; to select the default parameters, press Return. If you make a mistake selecting the choices, a "**" appears and that section of the program starts again; simply type in a correct choice.
6. Press the CapsLock key.
7. Type ATDT if you have a Hayes or Hayes-compatible modem.
8. Type the telephone number.
9. A message tells you when the dialing is complete.
10. When a message appears saying Connect, press Return. Type the log-in sequence.
11. As before, breaking the connection depends on the bulletin board.
12. To leave the terminal program, select "Exit Terminal" from the menu.

When you select the upload or download file options, a space appears for you to type a name for the file. Type an Amiga device name and the filename, such as df1:downfl.txt, where df1 is the Amiga BASIC term for an external drive attached to the computer and downfl.txt is the filename. As soon as you finish typing the filenames and press Return, the Amiga begins sending or receiving information. If you're sending more than one file, repeat the process for each one.

To upload or download a file, first make sure both you and the other computer are ready to send and receive. If you are going to upload a file, make sure the other computer has already named and opened a file so the Amiga has someplace to send the data. If you're going to download a file, select the download file option and name the file before telling the other computer to send the data. Failing to open and name the files first doesn't interrupt communications, it just means that the data disappears on the wires. A hint for long files: set the baud rate at 300. It may cost you more in terms of the telephone call, but there is less chance for a communications malfunction at the lower speed. Use the 300 setting for getting the *AmigaTerm* program. It is about 40K and will take about 45 minutes on the telephone, but you'll be sure to get it all.

THE AMIGA ON-LINE

Once your modem, telephone, and Amiga are connected, and you have some communications software, you're ready to put the Amiga on-line. For communications to commence, one computer has to be designated a "terminal" and the other designated the "host," because two hosts or two terminals cannot communicate. The technical reason for this requirement is that two hosts or terminals would try to send data using the same transmission frequencies on the telephone line, hopelessly mixing the data being sent from one to the other. Fortunately, you do not have to be concerned with determining which is which; the communications software does the job for you automatically.

The host/terminal concept is analogous to the connection of a printer to your Amiga. The Amiga is the host having computing capabilities, the printer simply receives data from the host. When two computers are connected, the signals pass over the serial port (the same one that attaches some printers to the Amiga), "tricking" the computers into seeing the telecommunication as identical to the printer connection. "Terminal emulation" is the term that describes designating one computer the terminal and the other the host. Knowledge of this concept is necessary when using software that asks questions like "Host echoing required?" — meaning that the host sends a copy of received data back to the Amiga's screen.

Many terminal-emulation programs require that the microcomputer act strictly as a terminal — no computing, processing, or getting files — while the communication is in progress. With the Amiga's multitasking abilities, however, you can run the terminal-emulation program simultaneously with other programs and computer functions.

Handshaking and Protocol

After you establish the proper host/terminal relationship, communications can begin. The two computers send messages to each other that describe exactly how the communications will occur, how data will be transmitted, which XON and XOFF characters will be correct, and so forth. This setup process, known as "handshaking," is strictly between the computers. The agreed upon communications process is called the protocol. The communications software completes the handshaking and establishes the protocol automatically.

The baud rate, parity, and other settings define the major portion of the protocol. If the computers have some incompatibilities, such as unequal baud rates, the handshaking will not occur and communications will not take place. Should that situation occur, changing the settings for the Amiga or other computer often rectifies the problem.

Following a successful handshaking, the real communications can begin. If you are sending data to someone else sitting in front of an Amiga computer, the screen may at first be blank or show a few messages identifying the status of communications (e.g., one message may be simply "communicating"). But, as you type data on the screen, it passes over the telephone lines and appears on the other computer's screen at almost the same time it appears on yours.

In addition to typed data, the Amiga can send data directly from files, or if the communications software has the capability, from multiple files in sequence. Furthermore, using the Amiga's multitasking capabilities, transmitted data can be simultaneously displayed on the screen, stored on a disk, and printed on a printer.

Troubleshooting Communications

Computer communications is still in its infancy and problems frequently occur. In fact, telecommunications *without* problems is considered unusual. Here are ten of the most common problems, but the list is by no means complete.

Communications will not begin. Usually the handshaking requirements are not being met — for instance, the baud rates are not the same or perhaps a modem's DIP switches need resetting. Of course, both computers must be ready to transmit and receive data before communications will begin. Something as ordinary as an unplugged modem has been known to cause hours of frustration. A loose-fitting rubber cup on an acoustic coupler modem may also prevent communications.

A connection will not occur if the telephone lines are busy or if the serial port normally used for the modem is running a printer. Likewise, the connection fails if the computers send signals that indicate they are both hosts or both terminals. In cases that require a password or specific log-in procedure, failure to type the proper password or to follow the log-in procedure usually causes the communications connection to wait. Occasionally, typing a wrong password "hangs up" the telephone.

Received information is unintelligible. Usually the options in the communications software are set incorrectly, but not bad enough to preclude handshaking or transmission. Stop the communications, change the settings so they match for the two communicating computers, then transmit the data again.

Some received information is readable, but other is unintelligible. Every so often a computer will not send or receive a bit. This is known as "dropping a bit" and is a technical shortcoming of computer communications. Although infrequent, and the intended target of parity checks, it still happens.

Sometimes a noisy telephone line is at fault. If that's the problem, whole sentences and paragraphs are normally incorrect; rarely are just one or two letters wrong.

Control characters or other extra characters can also cause the problem. For instance, formatting characters for text in a word-processing file may be interpreted by the receiving computer as other characters. The received text then appears garbled. Some communications software includes a special command just for this situation. Known as "Filter" or "Translation Table," the command catches the extra characters and filters them from the data. Another solution is to send writing as an ASCII (or AmigaDOS) file. ASCII files do not contain formatting control characters.

The format of the transmitted and received data differs. Some communications programs can set the Amiga's screen format to match the other computer's screen. Usually the setting is 40 or 80 columns to be displayed on the screen, although other settings are possible. Having both computers set at the same screen width helps maintain format consistency, especially for text.

Spreadsheets, charts, graphs, tables, and graphics, on the other hand, are often out of kilter, regardless of the format settings. This occurs because control characters used to create the data do not serve the same function on the receiving computer. Seeing such data in its original form usually takes some creative reconstruction and reformatting. These kinds of formatting problems can be difficult to solve.

Not all the transmitted data is received. If transmission is at higher baud rates and the flow control setting (XON/XOFF) is not operating, data is lost when the receiving computer's buffer is full. To rectify this problem, turn on the flow control. You may also run into peculiar problems that lose data. For example, data transmitted from disks with 8 storage sectors (common on the IBM PC) to an Amiga disk with 11 sectors might extend the left margin of the data beyond the screen, losing the beginning of each line. To fix the problem, reformat the Amiga disk to receive only 8 sectors.

Data appears only on one screen line. When this situation occurs, one line of data appears and is then overwritten by the next line of data. The rest of the screen stays blank. Almost always, this problem is due to the absence of a line-feed character at the end of each data line. Without a line feed, the receiving computer does not move down the screen to the next line. Most communications software has instructions for sending a line-feed character.

Data is printed on every other line of the screen. This problem is the opposite of the previous problem. If the sending computer is transmitting a line feed at the end of each line and the receiving computer is automatically inserting one too, the data is printed on every other screen line. The solution, obviously, is to turn off one of the two (but not both) sources of the line-feed character.

Typed data doesn't appear on the screen or each character appears twice. This is usually a problem with the combination of software settings for echoing and full/half duplex. Either the characters aren't being echoed or they are being echoed twice. Characters being echoed twice llooookk lliikkee tthhiiss. To rectify the problem, change the setting for either echoing or full/half duplex. If that doesn't work, see if the communications software differentiates between host echoing and local echoing and try those options.

The communications link abruptly disconnects in the middle of transmitting. This exasperating problem occurs all together too often and is usually of mysterious origin. Simple causes are: someone at either end of the communications picking up an extension telephone; Call Waiting clicking noise; other spurious line noise such as pops and crackles from lightning or other electrical disturbances.

Harder to diagnose are problems that have strictly electronic origins. For example, if XON/XOFF is turned on and the data being transmitted includes the characters the receiving computer recognizes as those that start and stop transmission (e.g., the control key followed by a *Q* or *S*), the computer may sense those symbols as being for flow control and turn off the transmission. Likewise, symbols in the data that the receiving computer treats as interrupts or pauses in the transmission may also cause the communications to end. Quite often these kinds of problems defy any reasonable attempt to solve them. Retransmission is normally the attempted recourse.

The receiving computer "freezes up." Although many computer manufacturers don't like to admit it, a number of machines just stop for no apparent reason in the middle of communications. This event is known as locking up or freezing the computer. Often the problem is that too much data is being sent at one time. For instance, if you're trying to transmit a book manuscript as one huge file, the receiving computer may not be able to process that amount of data at one time and will simply stop. (One computer notorious for this problem is the Xerox 850. According to Xerox's technical representatives the only solution is to "quickly turn the computer off and then back on again." Naturally all data transmitted and in the 850's RAM is lost.) The solution is to break the data being transmitted into smaller segments and transmit each one separately. Though more expensive for the telephone charges, at least the communications will be successful.

A FINAL WORD ABOUT TELECOMMUNICATIONS

Computer communications holds great promise for the future but currently is undergoing some rather trying growing pains. Ironically, part of the problem with today's telecommunicating is that telephone calls are so easy to make. The technical complexity and sophistication of telephone systems are completely masked each time you make or receive a call. You don't have to know that a call is being sent through a satellite 22,000 miles in space or that hairlike optical fibers or even laser beams are carrying electrical signals that eventually will become your voice again at the end of the line. Nor do you need to know that a call from New York City to Washington, D.C., may go by way of Cincinnati. As far as callers and receivers are concerned, a number is dialed and a conversation begins.

It wasn't always so. Making a call on a telephone network in the early 1900s required dialing the local switchboard and giving the operator the number to be called. The operator then placed the call and made all the necessary connections.

In terms of computers, communicating is about where the telephone system was in those early years except with two major differences: the existing, complicated telephone system is already a part of telecommunications networks, and the level of technical expertise required to understand the concepts of communications goes well beyond the level required by the early telephone pioneers. Nevertheless, pioneering is what communicating with personal computers is all about. As more networks are formed, the arcane jargon surrounding computer communications and the technical complexity of telecommunicating itself will give way to simplicity until computer-to-computer communications is as easy as an everyday telephone call. But until that time, telecommunicating will remain a new, evolving affair that, like the telephone, may profoundly affect day-to-day life. Learning the hows and whys of telecommunicating with the Amiga is one way to make the most of the communications changes that inevitably will take place in the future.

CHAPTER

11

Connecting with the World Outside

You have a lot of choices for making connections with your Amiga computer. You could simply make the connection with another single computer — a friend's, neighbor's, or business client's. Their computers do not have to be Amigas or even Commodore products. By following the instructions of your communications software manual, you should be able to make the linkup successfully, but be aware that getting this type of communication started can initially be full of frustrations (as the problems described in chapter 10 attest).

More common than just two computers talking together is connecting with a network. The whole idea behind computer communications is to give you access to many different sources of information, and to many other computer owners. The following are descriptions of some common networks.

MORE ABOUT ELECTRONIC BULLETIN BOARDS

Much more common than telecommunication between just two owners of personal computers is the electronic bulletin board. Electronic bulletin boards are computers on which anybody can leave messages, or read messages left by others. The sysop ("system operator") is an experienced computer user who dedicates a personal computer to transmitting or receiving messages over the telephone lines.

Connecting with a bulletin board is generally an easy process; many boards do not even require a password. You just make the telephone call, follow a few simple instructions to make the computer connection, and begin typing or reading messages. However, the connection procedures differ somewhat and some restrictions do apply. Some bulletin boards require that you type the Return key three times before the connection is complete; others, which use a ring-back procedure, require that you let the telephone ring once, hang up, and then call back within one minute. Because they run on a sysop's personal computer and home telephone line, electronic bulletin boards rarely operate seven days a week or twenty-four hours a day.

Another restriction is the type of information carried on a bulletin board. Some emphasize political discussions, others are for literary reviews and criticisms, Bible discussions, chess and other strategy games, for physicians to exchange medical information, and for kids to communicate with a computer. A few bulletin boards are for psychological counseling or even X-rated topics. Larger bulletin boards handle a number of separate subjects, called conferences, which are further broken down into topics. For example, one bulletin board has a conference for Pascal programming, and one of the Pascal topics is "Turbo" for the program *Turbo Pascal.*

The number of topics covered by bulletin boards is growing, as are the number now operating. Currently over 1,500 bulletin boards are active in the United States, with more being added daily.

How do you find out what's available? Two bulletin boards are devoted solely to keeping an up-to-date list of the bulletin boards. The two are Novation Modem Information located in Tarzana, California, telephone 213-881-6880 (Novation is a modem manufacturer) and On-Line Computer Telephone Directory in Kansas City, Kansas, telephone 913-649-1207. A spate of magazines such as *Link-Up*, *Computer Shopper*, and the *Computer Phone Book* are devoted to bulletin-board communications as well.

Due to software limitations, some bulletin boards can only accept certain types of computers; for example, many that use a variation of software known as the Remote Bulletin Board System, or RBBS for short, are for IBM computers only. Also, some only operate at certain baud rates (almost always 300 and 1,200, or both).

As the free *AmigaTerm* program demonstrates, one of the benefits of electronic bulletin boards is the wealth of free programs and data available on them. Typical free programs calculate income taxes, figure out investment risks and stock brokerage commissions, draw block letters and create color displays, play dice games, card games, chess, and bingo, write music, analyze IRA accounts, and do word processing. All for the price of a telephone call.

Another benefit is the expert answers you can receive to technical computer questions. Especially attractive to new computer owners, questions about operating a computer sent to the bulletin board will inevitably be answered by someone who experienced the same problem and found a solution for it. In this way, electronic bulletin boards are not only forums for discussions, they are becoming instrumental in furthering peoples' dexterity with operating computers.

A good bulletin board for getting Amiga questions answered is known as Bix. Run by *Byte* magazine, the Bix bulletin board's subscribers include professional programmers from Amiga and Commodore, and from many software companies, such as Electronic Arts, Inc. Some of the programmers are the authors of the software you use daily on your Amiga; others are developing new programs for it. If you have a question about the Amiga, or want some free advice, Bix is one good answer.

Bix carries two separate conferences, called Amiga and Amiga.user, for Amiga owners. The Amiga conference is generally more technical than the Amiga.user one. Bix costs a one-time registration fee of $39, or $25 if you are a *Byte* magazine subscriber. Hourly rates are $12 from 7:00 A.M. to 6:00 P.M. weekdays plus the telephone call. From 6:00 P.M. to 7:00 A.M. weekdays, and on weekends and holidays, the hourly fee is $9.

To connect with Bix, first get the local Bix number for your area. Bix maintains local lines in some major urban areas. For instance, in the San Francisco Bay area the number is 415-982-2151. Check the telephone directory for the number in your area or call the toll-free Bix Customer Service Line at 800-227-2983. (From New Hampshire or overseas call 603-924-7681, which is a toll call.) If you live in an area not served by a local Bix line, you'll have to go through Tymnet, a computer/telephone service. If you can't find a Tymnet local number, call Tymnet at 800-336-0149 or

Bix Customer Service. Tymnet charges are $6 per hour for the daytime, and $2 for nights, weekends, and holidays.

The second number to obtain is from your Visa or MasterCard. Bix bills you on either of these cards, and the first time you join Bix you have to give the card number. Thereafter, only your name is necessary to make the connection.

With both numbers firmly in hand, you're ready to set the computer for communications. You have two options for the communications software:

- Full duplex, 8-bit words, no parity, 1 stop bit, and either 300 or 1,200 baud.
- Full duplex, 7-bit words, even parity, 1 stop bit, and either 300 or 1,200 baud.

Start up your communications software and enter in one of the two options. Now call the local Bix or Tymnet number with your modem and step through the following screen instructions. The first three instructions are for Tymnet connections only. Skip them if you're calling a local Bix number.

The first request either asks for a "terminal identifier" or is garbled. Don't be alarmed at the garbled words — they occur when the baud setting is 1,200. Type the letter *a*. A message asks you to log into Tymnet.

Type the word *byteneti* and press Enter. A message asks for a password.

Type *mgh* and press Enter. A message asks for a log-in. (If the message fails to appear, press Enter again.)

Type *bix*. The Bix logo appears on the screen. If the logo fails to appear, press Enter. A message asks you to enter your name. (Don't be alarmed if the words or lines appear more than once on the screen. Extra echoing happens frequently.)

For your first connection with Bix, type *new* and press Enter. A series of prompts ask for your name, résumé (whatever you want to say about yourself), and other registration information. Upon completion of that data, you'll be taken through a quick tutorial to learn and use the system. A hint: be careful with your résumé information — you never know who will read it. Also, *Byte* magazine extracts some of the more informative messages and publishes them each month. You may (or may not) want your name in lights.

Later, to connect with Bix, type the name you've assigned yourself and press Enter. You can update or change your résumé, too (use the "show resume" command).

For all its benefits, Bix can be difficult to use for two reasons: first, its commands are ponderous and require a lot of typing; second, it has become so popular that the number of messages on it are getting too numerous to read. A better option may be your local Amiga owner/user group.

Many Amiga user groups maintain networks to answer members' questions, share information, and programs. Members may also meet regularly in person, although some conduct all their meetings solely on a bulletin board. Most are free, and because they are located in your immediate area, don't even cost for the phone call. Try contacting your local Amiga dealer for information about Amiga user groups. One particularly good reason for joining such a group is to trade a copy of your free *AmigaTerm* program (see the previous chapter) with someone who has the Lattice C compiler. Give that person two copies of *AmigaTerm* and get one of them back compiled for you.

INFORMATION SERVICES

Also known as on-line services or databases, information services are an outgrowth of business information networks previously available only to companies owning a large computer. For a stiff subscription fee, business subscribers can request data from the networks on any of the myriad of subjects covered, and then pay additional charges for the amount and type of information received. Up-to-the-minute stock quotes, news flashes, business transactions, and the latest scientific and medical information are subjects well covered by the information networks.

Although lucrative, the networks were not generating much income during nonbusiness hours, and the computers managing the networks were idle. But with the phenomenal growth in the numbers of people owning personal computers, especially among many of the same people who were using the networks through their companies, came the opportunity to open up the networks to other subscribers. Information networks began actively to solicit and offer their services to owners of personal computers. Acceptance of the concept began as a trickle but has today become a flood. The late 1970s saw the emergence of three networks that now dominate the information services. The three are Dow Jones News/Retrieval, CompuServe, and The Source.

How to Use Information Services

A hint to save you money: some of the information services charge you more for a 1,200 baud hook-up than for 300 baud. If you're downloading data from the service, the higher 1,200 charges may save you money if they offset the charges for the connection fee.

But if you're carrying on an active real-time dialogue with someone (called the CB mode on many services because it's like talking on a CB radio), the 1,200 charges are money wasted. Why? Because you have to type your conversation. No one can type at 1,200 baud (over 100 characters per second!), so there's no reason to pay for the higher rate. When you're planning a CB session, always set the baud rate at 300 or less.

Dow Jones News/Retrieval Service

Started for business in 1974 and opened to personal computers owners in 1977, the Dow Jones Service has over 100,000 subscribers and is the most popular information service. It offers stock market quotes, financial and investment information, news, weather, sports, movie reviews, a variety of business data, and encyclopedic information. Initial subscription costs $50; charges per use depend on time of day and data requested. Prime-time hours are business hours, 6:00 A.M. to 6:00 P.M. Monday through Friday. Charges during those hours range from $54 to $72 per hour. During nonbusiness hours, and on weekends and holidays, charges range from $9 to $54 per hour. You also pay for the telephone call to make the communication connection. Getting a quick stock market quote during nonbusiness hours usually runs around 25 to 50 cents. The service is closed daily from 4:00 A.M. to 6:00 A.M. (New York local time) to update the information bases.

Dow Jones and Company, Inc., P.O. Box 300, Princeton, NJ 08450; telephone 609-452-2000.

CompuServe

CompuServe began in 1970 as a data-processing company and is now a major provider of electronic information. Over 50,000 people now subscribe to the CompuServe information services. The available data is divided into four basic categories: Business and Finance, Home Services, Personal Computing, and Services for Professionals. These four are further subdivided into over 200 separate databases, such as shopping advertisements, banking services, stock quotes, bibliographies of computer articles, and travel-arrangement services. The Personal Computing section has an Amiga topic. A bulletin-board service and electronic mail capabilities are also available. Subscription costs $20 to $40 which includes five free hours of introduction to the service. Prime-time hours are 8:00 A.M. to 6:00 P.M. weekdays; the charge during prime time is $12.50 per hour. The charge at other times is $6 per hour. The costs of the telephone call are not included in the charges and must be paid separately. Additional charges are made for using specialized databases. CompuServe operates twenty-four hours a day, seven days a week.

CompuServe Inc., 5000 Arlington Centre Blvd., Columbus, OH 43220; telephone 614-457-8600.

The Source

The Source is an information service owned by the Reader's Digest Association. Originally oriented to the consumer with databases about home and leisure, sports, travel, and home catalog shopping, The Source is now courting business people. In addition to database services, The Source offers electronic mail, bulletin boarding, and computer conferencing between two or more subscribers. The subscription fee is $49.95 and the prime-time charge is $20.75 per hour. Prime time is 7:00 A.M. to 6:00 P.M. weekdays. From 6:00 P.M. until midnight and all day weekends and holidays, The Source costs $7.75 per hour. From midnight to 7:00 A.M. the charges are reduced to $5.75 per hour. Subscribers also pay a monthly minimum fee of $10. Currently over 40,000 people use The Source.

Source, Telecomputing Corp., 1616 Anderson Road, McLean, VA 22102; telephone 703-734-7500.

Other Information Services

Smaller information services abound. Usually these services are more specialized, such as academic bibliographic services, medical abstracting databases, and literary compilation services. Some are also beginning to offer the use of programs, such as word processing and spreadsheets for subscribers. This may be an inexpensive way to try out and evaluate different programs that you're considering for your own personal use.

Computer information services that serve localized audiences are also beginning to take hold. Services that provide local dining reviews, schedules of sports events, and other local news are available in some areas. Generally cheaper both for the telephone call and the use fees than the larger services, these local services are often an extension of a bulletin board.

ELECTRONIC MAIL

As its name implies, electronic mail is for sending letters over the telephone lines. Unlike on the bulletin boards where everyone can read your notes, electronic mail is sent only to the addressee's computer. Most of the information services offer electronic mail options and assign you an "address" or "electronic post box."

Although the telephone call necessary for electronic mail may be a bit more costly than mailing a standard paper letter, service is instantaneous and can be verified by the recipient. Also, if you're sending a lot of data, it can be stored directly on the other computer's disks or tapes without any retyping. By having a modem and software with auto-answer capabilities, you can set up the Amiga to receive electronic mail overnight. In the morning, mail sent to you will be noted on the screen and you can read the mail at your leisure.

At present, electronic mail is usually sent by people who have something to say to each other, but as computers become more ubiquitous, junk mail can't be far behind.

Glossary

Acoustic Coupler — An inexpensive type of modem that translates audible sounds from a telephone into a digital form that the Amiga can understand. Acoustic couplers typically have two rubber cups in which you place the telephone handset.

Active — The item or thing available for your instructions. The active window, for instance, is the one on which you can program or work. The term "current" is sometimes used as a synonym for active.

ADSR — An acronym for "Attack, Decay, Sustain, Release" that describes the shape of a sound wave, typically for a musical note, that produces a realistic sound.

Amiga BASIC — The "dialect" of BASIC available for the Amiga. Amiga BASIC has the same structure and many of the same commands as other versions of BASIC, however, it also has a number of commands specific to the Amiga. Many of the sound and graphics commands are for programming the Amiga's unique audio and visual capabilities.

Alert — A flashing warning message that appears at the top of the Amiga screen. Errors caused by system failures, or simply by an "out of memory" condition can produce an alert. If the problem is severe, the Amiga may reset itself causing a loss of all data in memory. However, the Amiga tries to assure that all disk memory is safe by initiating procedures to save disk files. Trying to respond to an alert before the disk procedures are complete may override them, therefore wait until all disk activity is completed before responding to an alert. See also *Guru Meditation*.

Animation Chip — A custom processing chip inside the Amiga. The animation chip assists the M68000 microprocessor by controlling: direct memory access requests, synchronization of video and sound, the output of other custom chips relative to the video output, and the movement of screen images. See also "Blitter" and "Copper."

Array — A group of variables, either numbers or characters. The individual variables in the array, known as elements, can be inputs to a program or process. For example, an array of numbers can be points on a waveshape that defines a particular tone for the Amiga sound system.

Aspect Ratio — The length-to-width ratio of the image on the screen. If screens were perfectly square, the ratio would be 1, but because screens are rectangular, images must be wider than they are high to look correct. The easiest image to illustrate aspect ratio is a circle. An incorrect aspect ratio produces an ellipse instead of a circle. Some graphics programs change the aspect ratio in order to create fantasy pictures. The Amiga BASIC CIRCLE command changes aspect ratio.

Assembler Language — A programming language that uses "words and phrases" that the Amiga can translate directly into byte values and binary 0s and 1s. Assembler language programs are specific to a particular computer and cannot be used on other machines. Programs written in assembler language are usually powerful and require a minimum of memory space, but effective use of assembler language requires a professional programming background. Assembler language is also known as a "low-level" language because it is the first step above machine language which is all 0s and 1s. Amiga BASIC, by contrast, is a "high-level" language.

Audio Chip — A custom processing chip that controls sound output as well as interrupts to the M68000 microprocessor. Having the interrupts handled in this fashion is one reason for the Amiga's multitasking capability.

Background — (1) The portion of the screen not devoted to text characters or graphics. That is, wherever text or graphics appear, they are in the "foreground," while the rest of the screen is the "background." The background color is usually different than the foreground. (If it isn't, you won't see anything except a blank, but colorful, screen.) (2) A process that is being run while you are doing something else. For instance, if a program is running some calculations while you are playing a video game, the calculations are being done in the background, or as a background process.

Bars — On the Amiga screen, bars are the horizontal or vertical symbols for moving an item, such as a window, to a different screen position. The Drag Bar on the top of a window moves (or drags) the entire window around the screen.

Batch File — A file that contains a set of commands, usually AmigaDOS commands, that are run as one batch when the file is executed. For example, you could set up a batch file that runs automatically every time you turn on the Amiga. The file might have the Amiga say good morning to you and play a little tune.

Baud Rate — The speed at which bits are transmitted from one computer to another. At lower speeds, baud rate is about the same as the number of bits transmitted per second. At higher speeds, several bits are added to the data stream to keep track of the transmission process and therefore baud rate and bits per second are not similar measures.

Bit Map — An area in the Amiga's memory that holds the data defining the colors of individual pixels on the screen.

Bit Plane — A sub-division of a bit map. A bit plane holds a single bit per pixel on the screen. By combining several bit planes for a single screen image, the Amiga can effectively assign more than one bit per pixel to produce a bit pattern. The bit patterns correspond to a color table and thus determines the actual colors of each pixel.

BIX — A bulletin board run by *Byte* Magazine. BIX has separate conferences for the Amiga owners and experts.

Blitter — A nickname for "bit-mapped block transfer." The Blitter is a section of the animation chip that controls the movement and positioning of images on the screen.

BOB — An acronym for Blitter Object. A BOB is a rectangular area of the screen that you can move to another position on the screen. The rectangular area might contain images of a missile or spaceship or some other figure that you want to move without having to re-draw it. Moving BOBs, however, usually requires programming in a lower level language than Amiga BASIC.

Bulletin Board — An "electronic meeting place" that you access with your Amiga on the telephone lines. Computers set up by individuals or businesses act as the central point for leaving messages, reading data, or carrying on conversations using computers.

"C" Language — A programming language designed for professional programmers, not for people just learning about computers (which is what BASIC is for). C programs must be run through a compiler program that translate them into machine language. C programs written for one computer may be usable on other computers.

Call — A BASIC command to access a program or subroutine. Used generically, call refers to the process of transferring the flow of a program from one process to another. For instance, to get an Amiga subroutine from its standard library of routines, you must call that subroutine into your program.

Characteristic Waveform — A waveshape that produces a sound typical of a particular musical instrument. For example, if you attach an oscilloscope to an instrument and play a note, the trace on the screen is a characteristic waveform.

Collisions — When two (or more) screen images bump into one another on the screen. You can program the Amiga to treat the collision as occurring, in which case the images appear to bounce off one another; or as not occurring and the images appear to pass each other, unobstructed and on different levels.

Color Register — A portion of the Amiga that stores the colors assigned to each bit pattern for a particular screen. Changing the colors of the color table will change the settings in the color register.

Color Table — A table that shows the colors that correspond to the bit patterns created by the bit planes. You can define the colors you want in the table and can change them even after an image is drawn. If you do, the image is re-colored to reflect the new values you assign to the table.

Command Line Interpreter (CLI) — An alternate mode for operating the Amiga. Instead of selecting the Intuition icons or menu items to activate Amiga processes, you type commands when in the CLI mode. CLI is similar to operating DOS on an IBM type of computer.

Compact Disk (CD) — A disk that stores information using laser images instead of magnetic recording heads. Compact disks may become a storage media for Amiga data, images, and audio.

Coordinates — Two numbers that indicate a position on the screen. The first number is the horizontal distance (the x-axis) from the left edge of the screen; the second number is the vertical distance (the y-axis) down from the top edge. Distances are measured in pixels on the screen.

Copper — A nickname for a co-processor on the Amiga animation chip. The copper controls the output of the graphics and sound chips in order to keep sound and graphics synchronized on the screen.

D to A Converter — A device that translates electronic signals from digital to analog (D to A) form. D to A converters typically change digital (0s and 1s) signals from a microprocessor into voltages that drive a television tube.

Device — A generic name for any part of the Amiga that can send input or receive output. The keyboard is a device, and so is the COM1: port that the Amiga uses for communicating with other computers.

Device Name — The specific name the Amiga assigns to a device. For example, SCRN: is the device name for the screen, and KYBD: is the device name for the keyboard. In programming, you use the device names to tell the Amiga where to send ouput or receive input. The colon after the name tells the Amiga it is a device name and not a disk name.

DF0: — The name Amiga assigns to its internal disk drive. Other drives added externally are respectively, DF1:, DF2:, and DF3:. A hard disk drive is named DH0:.

Dimension — A characteristic of an array that tells how many "sides" it has. A one-dimensional array is a single line list of elements, a two-dimensional array is a table of elements.

Directory — A grouping of files on a disk. By defining a directory you can store all related files together and access them as a single group. Typically you create a directory for a program, and store all files pertaining to that program within the directory. Directories are also called Drawers.

Disk Name — The Amiga refers to disks by a name assigned to them instead of by the drive holding a disk. For example, on an IBM type of computer you must refer to a drive for the computer to read a disk but on the Amiga you simply refer to the disk's name and the computer finds it regardless of drive.

Dotted Note — A music symbol that indicates to play a note half again as long as is normal. In the PLAY program in this book, dotted notes are shown as the note with a period following it.

Dragging — The process to move items on the Amiga screen. Put the pointer on the item, press and hold down the left mouse button, and move the mouse. The item follows the pointer on the screen. When you release the button the item stays at its new position.

Drawers — A icon that is a small desk drawer and indicates a directory of files. By selecting the drawer, you can see what files are in it.

Driver — A program that runs a peripheral device, usually a printer. The Amiga contains a number of printer drivers as part of its standard software.

Dual Playfield — Two separate screen images that appear to overlay one another. A common use of the top playfield is to create a black overlay except for a single round hole making it appear as if you're looking through a telescope. The scene that you see "through the telescope" is on the other playfield.

Duplex — Allowing telecommunications in two directions. With full duplex you can send and receive simultaneously.With half duplex you can send or receive, but not at the same time.

Echoing — Sending back characters to their point of origin. If you transmit data from the Amiga to some other computer, and that computer sends the same data back to your screen so you can see what was sent, that other computer is echoing the data.

Emulation — To imitate a computer or terminal. An emulator program causes a computer to take on the characteristics of another piece of equipment. In telecommunications, emulating a terminal is often necessary when connecting with another computer.

Envelope — The ADSR values that create a sound wave for a particular tone.

Execute — To start, or run, usually a program. Thus, for example, to execute a BASIC program, you type Run and press the Return key. The AmigaDOS command EXECUTE followed by a filename runs the commands in the file.

Expansion Bus — A large plug-in slot on the Amiga's left panel that provides complete access to the hardware inside the Amiga. Instead of removing the Amiga case and installing new boards, you can simply plug them into the expansion bus in daisy chain fashion. See "Open Architecture."

File — A collection of data stored together. In the Amiga, files created by programs are called Projects.

File Name — A name you assign to a file, and that you subsequently use to tell the Amiga to access the data in the file. The Amiga will then either show you the data on the screen, or run the commands within the file.

Flat — A musical tone one half step lower than a specified note. In the PLAY program in this book you specify flats with the minus sign (–) following the note, i.e., A – is A flat.

Foreground — (1) The portions of the screen that show characters and images; the rest of the screen is the background. For instance, the text on a screen is in the foreground and the rest of the screen is the background. The foreground and background are usually different colors. (2) A process occurring on the current, or active, window. Because the Amiga can run different processes concurrently, there needs to be some way of indicating the process running on the current window as opposed to the processes running on other windows. The current window process is said to be running in the foreground, and the other processes are said to be running in the background.

Gadgets — Symbols on a window that let you change the window's size, select which window is to be active, and close the window.

GEL — An acronym for Graphics Element. GELs are visual collections on the screen made up of sprites and BOBs. By defining items as "components" of a GEL you can then treat them collectively as a single unit instead of individually. This makes graphics programming more effective. GEL programming is usually not done in BASIC. Assembler language or the C language are more often used with GELs.

Gen-Lock Card — An add-on card that locks the Amiga's video signals in phase with a video cassette recorder. Attaching a gen-lock card to the Amiga and connecting it to a VCR will allow output from each machine to be directed to the other. For instance, you could add Amiga animation to films played on the VCR.

Graphics Chip — A custom processing chip that controls the Amiga's video output to the screen.

Guru Meditation — A term followed by a series of numbers that appear when a system error occurs on the Amiga. The numbers are for Amiga engineers, technicians, and software programmers to determine the cause of the error.

Handshaking — The automatic process by which two computers establish that a telecommunications link will be successful. Handshaking checks that baud rates are correct, flow control options are on or off, and so forth. Once handshaking is complete, telecommunications can begin.

Harmonic — A musical tone that is an exact multiple of the main frequency of an instrument. Having harmonics gives an instrument's tone richness.

Heap Memory — A portion of the Amiga memory that is for the values of such things as static variables.

Host — The computer controlling communications in a network. When two computers are connected, one usually has to emulate a terminal and the other operates as the computer. That latter one is said to be the host. On large networks involving "mainframe" computers, the host is always the mainframe.

Icon — A small symbol on the Amiga's screen that represents an item or process that you can select. For example, the icon of a trashcan is the symbol for discarding information. Icons let you operate the Amiga without having to type words or phrases to get a job done.

Interlace — The way a TV screen's scan lines are arranged. If alternating scan lines overlap they are interlaced. If they don't (which is normal on standard TV images) the lines are non-interlaced. Interlacing produces a higher resolution image, but may cause flickering.

Intuition — The visual means to operate the Amiga. Intuition is the "icon-based" system by which you select the icons to read files, start programs, and so on. The alternative way to operate the Amiga is with the Command Line Interpreter (CLI).

Invoke — A somewhat ambiguous term meaning to start or call a programming function. For example, to invoke a subroutine is to call it into a program and start it running.

Kickstart Disk — The disk you insert into the Amiga to start the computer's operations. The Kickstart loads operating software and includes routines to check the condition and operating status of the internal hardware.

LAN — An acronym for local area network. LANs are a rapidly growing feature of computers. Typically used in business settings, LANs connect individual computers with wires or other direct attachments such as fiber optics. LANs allow computers to share files, programs, printers, and other computing resources.

Menu — A list of options presented on the Amiga screen. Selecting one of the options may produce a list of other options in a "pulldown" menu (that appears directly under the selected option).

MIDI — An acronym for Musical Instrument Digital Interface. A MIDI is a piece of electronic equipment that you attach to the Amiga. Then you can connect an electronic instrument, usually a synthesizer, to the Amiga and program the instrument directly from the computer.

Modem — A term derived from the words Modulate/Demodulate. A modem is a device that translates digital 0s and 1s into audible tones for a telephone. By connecting a modem to the Amiga, you can send data to other computers over the telephone lines.

Multitasking — Working on more than one computing job simultaneously. Because of its design with separate custom chips and a high speed microprocessor, the Amiga can process multiple jobs at the same time without any appreciable slowing down of any of them.

Narrator Device — The program that contains the rules for the Amiga's speech synthesis capability. The Narrator Device is stored on the Workbench disk.

Object — Screen images that the Amiga uses specifically for animation. Objects have size, shape, and color and are saved with individual names. Two similar objects are possible with the Amiga: sprites and BOBs. You must select one of these two types before defining an object further.

Octave — An interval of eight musical notes (as in *do ra me fa so la ti do*). In programming Amiga music you specify the octave as one of the elements of the tune.

Open Architecture — The concept of having the internal hardware of the Amiga (or any other computer) accessible for adding other circuit boards or supplementary pieces of equipment, such as a hard disk. Open architecture used to mean being able to physically take the cover off a computer in order to add boards. However, with the development of the Amiga's 60 pin expansion port that allows complete access to the hardware without dismantling the machine, open architecture now means simply having complete electronic access.

Output Window — The window on the Amiga screen on which you see output from a program or job. You can direct output among different windows on the screen at the same time.

Paint — (1) A synonym for filling a screen object with a color. (2) A programming jargon term for refreshing the screen. That is, painting the screen occurs each time the TV beam sweeps over the entire screen once.

Palette — A synonym for a color table. The palette is the colors available for coloring images. The number of colors on the palette can vary depending on the number of bit planes assigned to the screen.

Parity — A procedure to check the accurate transmission of data over the telephone lines or on a network. By adding an extra bit to each byte, the transmitting and receiving computers can add the number of bits sent and received to ensure that the answer is always even or odd. If the answer deviates, the computers know a transmission error has occurred and that data has been lost or garbled. You specify the type of parity checking when establishing communications.

Pause — A time interval in music when no sound is played. In the PLAY program in this book, the P symbol indicates a pause. Music playing on a particular voice is still, however, music playing on other voices can continue.

Pathname — A description of the location of a file by specifying a disk drive and directories. For example, if a file named TESTER.BAT is on the disk in the Amiga's internal drive, within a directory named STARTUP, the pathname would be DF0:STARTUP/TESTER.BAT.

Phonemes — A basic sound of a spoken language. Just as you can divide words into syllables, you can divide syllables into phonemes. For instance, the one syllable word "bat" consists of phonemes for the sounds of "b", "ah", and "t".

Pixel — A nickname for "picture element." A pixel is a dot on the screen that the computer can reference and color. Each pixel is actually one spot produced by the sweep of the TV guns' beam. The more pixels on the screen, the higher the image resolution.

PLAY Program — A program in this book that lets you program music using the standard note symbols (A, B, C, D, E, F, G). The PLAY program duplicates many of the functions of the PLAY statement found in Advanced BASIC used on many other computers.

Preferences — An icon that appears on the Workbench window. Selecting the Preferences icon lets you select customizing features for how you want to operate the Amiga.

Projects — The generic name for Amiga files that you store in drawers (directories) on disks.

Protocol — The agreed-upon communications parameters when two computers are sharing data. The protocols are such things as the baud rate, parity checking, and flow control. In LANs, protocols are more complex and produce a stream of characters added to the data so the host computer knows how to interpret data from computers on the network emulating terminals.

Radian — A mathematical measure of an arc of a circle. Radians are expressed as units of pi. You use radians when defining an arc with the Amiga BASIC CIRCLE command.

Refresh Rate — The rate at which the TV beam sweeps over the screen one complete time. Normal refresh rate is 60 complete scans of the entire screen per second.

Requesters — A question message that appears on the Amiga screen. Some requesters require that you identify disks necessary to answer the question. A Yes/No choice with the requester answers it.

RGB — An acronym for Red-Green-Blue monitor. An RGB monitor is a high quality TV set similar to studio monitors. By attaching an RGB to the Amiga, you can see extremely sharp and clear graphics.

Sharp — A musical tone one half step higher than a specified note. In the PLAY program in this book you specify sharps with the plus sign (+) following the note, i.e., C+ is C sharp.

Sine Wave — A smooth waveshape that translates into a clear tone from the Amiga's sound system.

Sound Table — Information in the Amiga about the volume, voice, and waveshape of sounds to be produced on the sound system.

Sprite — An animation object, or image, on the Amiga screen. Before drawing an animation image, the Amiga requires that you indicate whether the image is to be a sprite or a BOB. Sprites can move faster than BOBs, but generally allow fewer colors. There are other differences between them as well.

Stack Memory — A portion of the Amiga's memory used to keep track of parameters and addresses used in programs.

Start Bit — A bit that signifies the next byte is to be sent when two computers are telecommunicating.

Stop Element — The period of time a computer waits during telecommunications before receiving the next byte of data.

Strings — A sequence of letters or numbers that the Amiga treats as data, not as commands. For example, if you define a field for a variable, such as a zip code, the various zip codes represent the string variables that belong in the field.

Subdirectory — A directory within a directory.

Sweep — The action of the TV beam as it scans across and down the screen.

Synthesized Speech — The Amiga's capability to duplicate spoken words and phrases over its sound system.

SYSOP — A nickname for "system operator," the person who sets up and maintains an electronic bulletin board. SYSOPs are dedicated individuals who use their own computers as the network host.

Tempo — The speed at which musical notes are played. As part of the PLAY program in this book, you can set the tempo for Amiga music.

Tools — A synonym for Amiga programs. Tools have their own icons on the Amiga screen that you can select. Selecting a tool icon starts the program.

Trashcan — An icon representing a procedure to discard or delete files. By selecting the Trashcan and a file, the file is deleted from its disk. Under certain conditions you can retrieve discarded files from the Trashcan.

Turtle Graphics — A generic name for a type of graphics program that uses a small image to draw images on the screen. An outgrowth of the Logo programming language, the image was originally a turtle, which itself was adapted from a small plexiglass domed device that rolled on the floor drawing pictures in response to computer commands. Turtle graphics drawing programs are often used in elementary schools to introduce children to programming and using computers.

User Interface — The screen procedures you use to get the computer to do something. The Intuition screen is one user interface on the Amiga, the Command Line Interface (CLI) is another. In programs, the way you select menus or options constitutes that program's user interface.

Variables — An element of data, either numeric or text, that can vary in a program. By giving variables a name, you can refer to the name instead of each individual element. Thus, for instance, having a table of values as part of some computation means you only have to reference the table name in order to get the program to use the elements one at a time and run the computations.

Virtual — A generic term for creating the appearance of more capabilities than are physically present on the computer. For example, a virtual terminal uses a portion of memory that makes it appear as if one of the Amiga's windows is actually a terminal sending data to the rest of the machine. Other items can be "virtual," such as virtual disks, and virtual storage.

Voice — One of the Amiga's four sound channels, numbered 0, 1, 2, and 3. You can direct which voice is to receive sounds to play.

Waveshape — The graphic representation of a sound wave for a particular tone. The waveshape replicates the actual sound wave as it travels through the air to your ear.

Window — A division of the Amiga's screen which acts as a separate entity. You can run one program in one window and another program in another window at the same time. Many windows can be on the screen simultaneously.

Workbench — A disk that reveals the Intuition system and includes the Clock, Trashcan, and Preferences icons.

XON/XOFF — A flow control paramenter for computer communications. The X stands for "transmission" and the XON/XOFF determines how the computers will start and stop communications. XON/XOFF is usually used in the case where the receiving computer cannot process the incoming data fast enough and begins to overload. The receiving computer sends a character to the other computer to stop the transmission (XOFF) until the data already received can be processed. Once the data is processed and enough memory is clear to receive more transmitted data, the receiving computer sends a character to re-start communications (XON).

Index

A State-Of-The-Art Guide For The World's Most Exciting Microcomputer

Hobbyists and business professionals alike have hailed the Commodore Amiga as a major breakthrough in the microcomputer world. Offering advanced graphic and audio capabilities, as well as a unique software/hardware design, the Amiga has set the standard for years to come.

The Amiga User's Guide to Graphics, Sound and Telecommunications is a clearly written, yet detailed tour of this technically innovative system—from Amiga's basic functions to its most sophisticated and exciting features. Whether you're an expert or novice, you'll find this book an invaluable tool for getting the most out of your Amiga. Included are comprehensive, screen-by-screen tours of these exciting features:

- **Musical Functions**: Use Amiga's unique and enhanced Amiga BASIC routine to program/compose musical scores in stereo on the machine's own keyboard.
- **Amiga's Voice**: Teach your Amiga to talk in a variety of tones, pitches, and even in a foreign language!
- **Graphics**: Learn to draw simple or complex images and get the most out of Amiga's color palette of more than 4,000 brilliant shades and hues.
- **Animation Capabilities**: Create and animate stunning action figures using Amiga BASIC's extensive list of special animation commands.
- **On-Line Communications**: Visit other Amiga users via modem and telephone lines. Included are instructions for accessing user bulletin boards and a *free* communication software program to make the job easier!
- *All this and much more!*

David Myers, award-winning writer, former college instructor and systems analyst at Stanford Research Institute, is author of *Personal Computer Buyer's Guide* and *VisiWord: Word Processing for You and Your Business*.

Also in the Bantam Amiga Library:
The AmigaDOS Manual:
Commodore-Amiga's official DOS documentation.

51895

9 780553 342833

N 0-553-34283-5>1895

34283-5 ■ IN U.S. $18.95 (IN CANADA $23.95) ■ BANTAM COMPUTER BOOKS