COMPUTE!'s

AMIGA

PROGRAMMER'S GUIDE

Edited by Stephen Levy

The comprehensive guide to programming the Amiga personal computer. Includes dozens of example programs, details on how to access advanced features, and a thorough explanation of BASIC.

A COMPUTE! Books Publication

$16.95

COMPUTE!'s AMIGA Programmer's Guide

Edited by Stephen Levy

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

The following articles were originally published in *COMPUTE!* magazine, copyright 1986:

"Getting Started with AmigaDOS," originally titled "Introduction to AmigaDOS" (January and February); "AmigaDOS Batch Files" (April).

"AmigaDOS Command Summary" was originally published as part of *COMPUTE!'s AmigaDOS Reference Guide*, COMPUTE! Publications, Inc., copyright 1986.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-028-9

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc. Lattice C is a trademark of Lattice, Inc.

Contents

Foreword

Whether you're programming your Amiga in BASIC, C, or machine language, *COMPUTE!'s Amiga Programmer's Guide* is the reference you'll need. The nine experienced programmers and writers who have contributed to this volume, explain how to program and use the power of the Amiga in the clear and concise style that has become the hallmark of COMPUTE! Publications.

Amiga BASIC, perhaps the most advanced BASIC available for any personal computer, includes commands that let a programmer access the powerful features of the Amiga. C. Regena has put together a complete reference section of Amiga BASIC commands, including sample programs that clearly illustrate how to program the Amiga.

Chapters 3 and 4 are AmigaDOS tutorial. Author Charles Brannon clearly shows you how to use the most popular AmigaDOS commands. We've also included a complete AmigaDOS command summary.

In chapter 5, Sheldon Leemon illustrates how to exploit the powerful graphics capabilities of the Amiga. Using program examples, you'll learn how to use "graphics.library" to produce the graphics you want.

The Amiga can produce sound that simulates the human voice. Philip Nelson, teaches you, in plain English, how to write programs using Amiga sound. Among the programs included in the sound chapter are a waveform editor, a phoneme builder, a speech experimenter, and a machine language sound generator.

Maybe you'd like to program in C or 68000 machine language. You'll find what you need to get started, including a variety of C and machine language program examples and appendices of selected Intuition, Kernel, fast floating point, and DOS library routines.

With its thorough introduction to the Amiga, easy-to-use reference charts, and clear, concise explanations of BASIC, C, and machine language, *COMPUTE!'s Amiga Programmer's Guide* is the one book every Amiga programmer should own.

About the Authors

Charles Brannon has been writing and programming for over six years. As program editor for COMPUTE! Publications, he has assisted in the development of COMPUTE!'s programming department and introduced the unique program typing aids, "The Automatic Proofreader" and "MLX," to the readers of COMPUTE! publications. Charles is also the creator of the bestselling word processor *SpeedScript* and coauthor of COMPUTE!'s *Advanced Amiga BASIC*.

Sheldon Leemon is a free-lance writer based in Michigan. He has authored or coauthored four books, including *COMPUTE!'s AmigaDOS Reference Guide* (with Arlan Levitan) and *Mapping the Commodore 64*. He is currently writing a fifth book, *Inside Amiga Graphics*. His work has appeared in numerous magazines.

Arlan R. Levitan is a staff regional systems engineer for Amdahl Corporation and telecomputing columnist for *COMPUTE!* magazine. He is the coauthor (with Sheldon Leemon) of three books including *COMPUTE!'s AmigaDOS Reference Guide.*

Dan McNeill is a free-lance writer based in San Francisco and Los Angeles. He is author of the book, *COMPUTE!'s Beginner's Guide to the Amiga,* and coauthor of *The Apple IIc: Your First Computer.* His work has appeared in numerous magazines.

Christopher D. Metcalf is a student at Yale University. He has authored numerous articles and programs for COMPUTE! Publications and has coauthored the book (with Marc Sugiyama) *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC and PCjr.*

Philip I. Nelson, assistant editor of *COMPUTE!* magazine, has written a number of articles on programming sound. He is also coauthor of the books *COMPUTE!'s ST Programmer's Guide* and *COMPUTE!'s 128 Programmer's Guide.*

C. Regena has a monthly column in *COMPUTE!* magazine and has written a number of books including *Elementary Amiga BASIC* and the bestselling *Programmer's Reference Guide to the TI-99/4A.*

Marc B. Sugiyama is a student at Harvey Mudd College in California. He wrote the very popular game "Zuider Zee" for the Commodore 64 and coauthored the book (with Chris Metcalf) *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC and PCjr.*

Tim Victor, editorial programmer for COMPUTE! Publications, has authored many sophisticated programs for various microcomputers, including "Apple Superfont." He is also coauthor of the books *COMPUTE!'s ST Programmer's Guide* and *COMPUTE!'s 128 Programmer's Guide.*

Chapter 1

Introducing the Amiga

Dan McNeill

Introducing the Amiga

Dan McNeill

The Amiga is the computer Merlin might have owned. It creates vibrant images and breathes life into them with sound. Indeed, it is perhaps the first computer to approach the mesmerizing audiovisual power of television. You can sit before it entranced by something as simple as a bouncing ball.

Speed. The Amiga is one of the swiftest personal computers available." Not only does its 68000 microprocessor work at 7.16 megahertz, but it has three extra chips which shoulder many tasks independently and boost its velocity.

Graphics. The Amiga's screen offers not just 640 × 400 resolution and 4096 colors, but a full graphics environment, with sprites, playfields, multiple windows, and bitmapped animation. These endowments are leading to extraordinary games and graphics and brilliant video in general.

Audio. The Amiga has four independent audio channels, with which it can synthesize almost any sound imaginable. It can imitate musical instruments, create its own sounds, and play along with you as an accompanist. The Amiga also has built-in voice synthesis, and making it speak is easy.

Multitasking. The Amiga can run several programs at once. Each program gets its own window on the screen and acts as if it is the chief and sole glory of the computer. You can compare two documents, write from notes, or keep different projects percolating at the same time. You can also control the computer, printer, and modem simultaneously, and thus print out one document while transmitting another and writing a third. The Amiga performs multitasking with ease.

Emulation. A $99 *Transformer* program can emulate the IBM PC; hence, the Amiga can run its vast library of business programs.

Expandability. The Amiga is fully expandable. Its memory expansion port accepts a simple memory upgrade which doubles the Amiga's RAM from 256K to 512K. It also has an expansion bus, where add-on devices hook into the basic circuitry of the machine. Expandability is particularly significant with the Amiga. Its graphics and audio strengths mean it will

become the tool of specialists, who will devise novel hardware for it. Such ingenuity has a trickle-down effect. Devices are invented for the space race and wind up on your kitchen drainboard. Hence, the Amiga may lead to new and fascinating retail products.

A quick tour of this computer must begin somewhere, and we might as well start with the first sight: the console's exterior. We'll then move on to the interface and after that to the basic hardware. Finally, we'll come full circle and look more closely at the Amiga's special powers in business, video, and audio.

The Facade

The Amiga may be the most deceptive-looking personal computer on the market. It consists of a plain, off-white console about the size of a stereo. Most of the external action takes place on the four sides of the console, each a strip somewhat narrower than a bumper sticker, and of these the front and back panels are the busiest.

The front panel. The front panel is the part you become most familiar with. A chromatic checkmark, the Amiga's hallmark, floats in the upper left corner, and next to it, the legend AMIGA itself. A pair of vertical lines trisects the panel, and a long horizontal indentation parallels the bottom. The middle third detaches to reveal the memory expansion bus. On the right third is the disk drive, the scene of much ado. The drive has a squarish recessed area to ease disk handling. Just below it to the right is the rectilinear disk-eject button.

The left panel. The left panel is relatively empty. It contains little but the on/off rocker switch, located near the front.

The right panel. The right panel boasts the two hand controller ports for various input devices you move around by hand. The mouse plugs in here, as do the paddle, joystick, light pen, digitizer pad, and optical scanner. The expansion bus is located here, too, beneath a removable plastic flap.

The rear panel. The rear panel has nine ports, where cables lodge like ships docked at a city (see Figure 1-1). They are, from left to right:

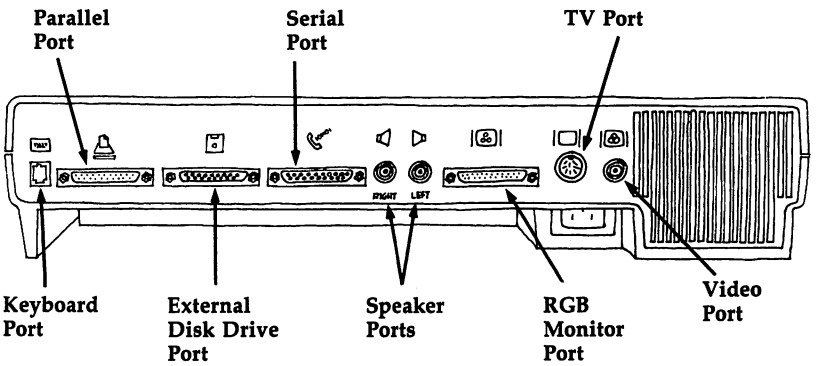
Keyboard port. This connector resembles a phone jack and accepts the keyboard cable with a brisk and easy click.

Parallel port. The D-type parallel port generally links up with the printer or the color plotter.

External disk drive port. This port allows a second disk drive to be added, which reduces the amount of disk swapping.

Serial port. Some printers and most modems use serial transmission rather than parallel, and this port handles them.

Figure 1-1. Rear Panel of the Amiga



The serial port is female and the parallel port male to avoid accidental mixups.

Speaker ports. These two jacks attach to the speaker in the Amiga monitor or to hi-fi speakers and let you bask in the computer's sound capacities.

RGB monitor port. This port links the Amiga to the RGB monitor. The icon shows a TV tube with three overlapping circles, emblematic of red, green, and blue.

TV port. This port sends signals to the TV set. You won't confuse it with either the horizontal RGB port on its left or the nozzlelike video jack on the right, for it is the only port that is circular. It takes several pins around its circumference and one at the middle.

Video port. This port connects to the leftover video miscellany: the monochrome and color composite monitors.

Bottom. Underneath the front of the console is room for the keyboard to slide in and out, a thoughtful touch that yields instant desk space if, say, you want to jot down a handwritten note or sign a letter. The genlock interface also fits beneath the console in the rear. The underside of the console

has vents which allow air to enter the computer and cool it while it works. As always, open vents are critical. Block them while the computer is on, and it may overheat and fail.

Top. The monitor normally goes atop the console, which can hold up to 40 pounds. This arrangement leaves slender shoulders on either side, upon which you can stack such peripherals as hard disk and modem, thus reducing the system's footprint.

Esthetically, the exterior very much resembles the IBM PC: solid and unprepossessing, not ugly, but no swan either. It scarcely matters. Once you turn the Amiga on, its appearance has little importance.

Entering the Amiga

You progress into the Amiga in a series of steps, all very simple. First, you turn on the monitor, then the computer. The fan begins to hum, and pale blue light floods the screen. Soon the little startup tune sounds, and the screen displays a large hand feeding Kickstart into the disk drive.

The Kickstart disk contains Intuition, the Amiga's operating system. Commodore decided to place Intuition on disk rather than ROM chips in order to make upgrades easily available, and indeed Kickstart 1.1 appeared within a few months of launch. This arrangement also lets you write your own operating system if you want to turn the Amiga into, say, a special-purpose graphics machine.

The Amiga loads the 256K Kickstart into a special, write-protected area of RAM. During this process, the red disk-use light stays on, and pressing the disk-eject button at this time can destroy the Kickstart disk. Unfortunately, the light sometimes turns off briefly in the midst of loading programs, so it's a good idea to wait for some other sign that the procedure is complete. Here, the screen will display a hand inserting the Workbench, and you can then press the disk-eject button and pull Kickstart out of the drive.

The next step depends on the software. It may have the Workbench or AmigaDOS already on it, in which case you ignore the onscreen cue and simply slip it in. Otherwise, you insert the Workbench disk and set the Amiga up for further tasks.

The Interface: Portal of the Amiga

The Workbench reveals the Amiga's interface, the arena where you and it conduct basic interaction. A computer's interface is in many ways its style, its personality. The Amiga's, like the Macintosh's, is a genial host, taking over the burden of communications and constantly proffering its resources so that you can select them at your pleasure.

The interface is also the gateway to the Amiga. It is an elaborate portal, full of important entry information as well as little aids which you can exploit or not at your pleasure.

Let's look at its fundamental elements.

The crux of the interface is the *pointer,* which is rather like a magic wand. You move it freely about the screen with the mouse. When it's over an item, you can click a mouse button and the item responds as if a magic wand had touched it. There is instant action. Touched items open up, fold down, or cause myriad changes, and you thereby gain mastery over the machine.

There are three kinds of items that react to the pointer's touch: *icons, windows,* and *menus.*

Icons. Icons are small pictures that act as doorways into various parts of the system. They come in four types: disks, drawers, tools, and projects. Disk and drawer icons open up onto further icons, while tool and project icons unveil actual work areas. The first two are containers for the second two, and the second two are the reason people buy the Amiga.

Disks. Disk icons are easily recognizable, as they resemble microfloppy disks.

Drawers. Drawers are repositories for tools and projects, and they usually look like desk drawers.

Tools. "Tools" is Amiga-ese for programs, like *Graphicraft* or the Clock. Most tools let you create pictures or documents, and their icons can look like anything a programmer can imagine.

Projects. "Projects" are files created by programs. For instance, a painting done with *Deluxe Video Construction Set* is a project.

When the Workbench interface comes up, there is a single disk icon in the upper right corner, labeled "Workbench." This icon is the first entranceway. Place the pointer over it, double-click the left button of the mouse, and a window appears with more icons inside it. You've moved into a foyer.

The Workbench window reveals a number of new icons. On the left are four drawers. Open a drawer—again, you double-click it—and another window appears, with more icons in it. You essentially move from the foyer to a smaller anteroom. If you click the Utilities drawer, for instance, a window with the Notepad icon appears.

The Notepad is a tool, one of the ultimate destinations in the system. Double-click the Notepad icon, and the Notepad window opens up. Here, you can jot down random thoughts and memos to yourself. You have reached the equivalent of a room, and you have no more doors to enter.

The Workbench has three other icons. One is the Clock. If you click it, a ticking clock appears onscreen, giving you the time. Another is the Trashcan. To erase a project you have no further use for, drag it over to the Trashcan. The Trashcan is a kind of drawer, and also a kind of limbo. It holds onto the file in case you want to recall it, since deleting a file is an act of consequence. Once you select Empty Trash from the Disk menu, however, there is no return.

The third icon is Preferences, which depicts a question mark over the front panel of the Amiga. Double-click it, and you enter the central command post of the Amiga. Here, you can dictate clock time, date, text size, mouse speed, double-click speed, Workbench colors, and numerous other matters. Preferences is actually a series of three screens layered over one another. Normally, you see only the first, the one most often used. But click Change Printer and the second screen appears. It allows you to set printer variables, an important function you cannot ignore. The second screen is also your entryway into the third. Click Graphic Select and you leaf down to the final screen, which lets you alter graphics on their way to the printer.

Icons do not exist solely to be opened. Instead of double-clicking an icon, you can click it just once. It will darken. You have selected it and can now perform a variety of operations upon it.

For instance, you can relocate the icon on the screen. Press the left button down and move the mouse around. When you get it where you want it, release the button and the icon will hop over. Moving icons can change their status, as it does when you drag them to the Trashcan. You can also shift

them in and out of drawers by dragging them from one window to another.

As we'll see below, selecting also lets you choose which icons the menu commands will act on. To deselect, click the pointer again somewhere outside the icon.

Windows. Windows are simply opened icons. They pervade the interface. When you click the Workbench icon, for instance, its contents appear in a window, and when you click drawers, more windows fly open. Indeed, windows are a highlight of the Amiga, and the computer can have at least 50 of them onscreen at once. That many is chaotic, but the power is there if you want it.

Often, with several windows open, you'll have to choose which of them is to be active. You do so by simply clicking the left mouse button once on the window you want. It's just like selecting icons and means the same thing: The file is ready to be acted on. You'll recognize inactive windows by their ghostly contours.

The pointer lets you control windows in another way, by clicking little symbols on them which trigger big changes. These symbols come in two types. The tiny, boxlike ones are gadgets, and the long, slender ones are bars.

Windows can have up to four different gadgets, as does the Workbench window. They are the close gadget, the back and front gadgets, and the sizing gadget (see Figure 1-2).

The close gadget. The close gadget is the square with the dot inside in the upper left corner of the window. Click it, and the window vanishes back into its icon. This operation is the reverse of double-clicking the icon.

The back and front gadgets. In the upper right corner are two gadgets which look very similar: the back and front gadgets. They shuttle windows behind or before each other on the screen, and you'll need them, because sometimes the screen will look like a wild stack of pages. The back gadget shows a white box behind a dark one, and the front gadget shows it in front.

The sizing gadget. The sizing gadget sits in the lower right corner of the window and shows a specklike box attached to a larger, vertical rectangle. It's meant as a before-and-after shot, since the sizing gadget lets you turn a small window into a larger one or into any size window you want. You place the

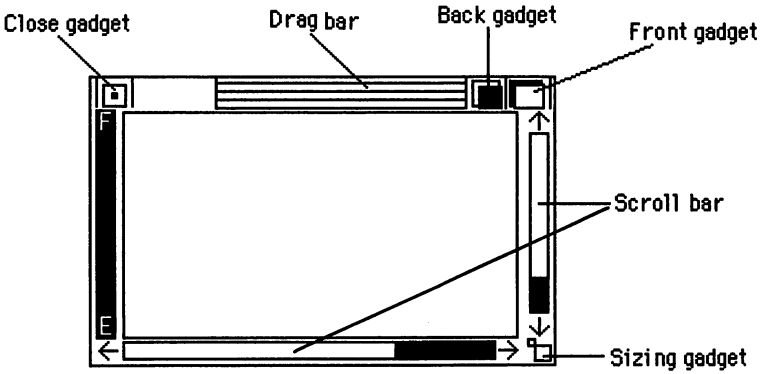
pointer on the gadget, hold down the left mouse button, and pull the mouse. The window expands or contracts until its new size pleases you, and then you release.

Windows can also have two different bars. One lets you shift the entire window about the screen like a hockey puck, while the other lets you move contents within the window:

The drag bar. The drag bar is the set of parallel lines at the top of the window. Point to it, hold the left mouse button down, and pull the mouse around. The window will follow. The drag bar lets you position windows on the screen.

The scroll bars. Contents of a window are not limited to window size. They can be much larger. In such cases, the window acts like a viewfinder looking down on a roll of microfilm. And the scroll bars on the right and bottom let you move the contents. Click on an arrow at either end of a scroll bar, and you'll shift half a window. You'll still have part of the old contents to give you bearings, but you'll enter new territory as well.

Figure 1-2. Controlling a Window



Menus. There is one final element to the Workbench triad, and it's crucial. It is the menu. All along, the screen has displayed a title bar across the top, explaining that you are in Workbench version whatever and have so many thousand units of memory free. If you press the right button of the mouse, the title bar changes at once into a menu bar with a string of menu titles on it. Workbench has the menu titles Workbench, Disk, and Special.

Icons and windows let you open, close, move, and delete files, yet many more specialized tasks remain, and menus let you perform them. They are simply lists of words and can thus be very flexible. Normally, menus are hidden to give you more screen space. Place the pointer over the menu title, and the menu unrolls, revealing a series of menu items. You drag the pointer down to the item of choice and release the right button. The Amiga then executes the task upon whatever you have selected.

Some menus have submenus appended to them. They flash out on the right as you pull the pointer down the menu. To select an item with a submenu, you shift the pointer over to the submenu and release at the chosen subitem.

Menu commands in applications can do almost anything, but most of those in the Workbench involve disk management. If you move an icon to the Trashcan and select Empty Trash, for instance, the disk drive whirs and the Amiga deletes part of the disk. The Workbench is thus an interface to the Amiga's disk operating system, AmigaDOS. The Workbench isn't AmigaDOS per se, only one pipeline to it. In fact, AmigaDOS also has a more direct connection: the Command Line Interface, or CLI. The CLI resembles CP/M and MS-DOS in that it requires explicit, typed commands, yet it gets you right into AmigaDOS itself and offers much greater power. You'll examine the CLI extensively in Chapter 3 and the Appendices.

We've now seen the exterior of the console and the interface, essentially the exterior of the operating system. Let's turn now to the hardware devices themselves—input, memory, processors, output.

Input Devices

The Amiga can use all the input devices of other personal computers, and its special capacities suit it for many new kinds. The basic input devices are, of course, the mouse and keyboard, but there are numerous special-purpose ones for games, graphics, and audio.

The mouse. The mouse takes care of basic screen control. The Amiga's mouse is about the size of a cigarette pack, though its beveled top—like the first facets of a stone in a jeweler's workshop—gives it a distinctive look. It also has two gray bars on top, called buttons. The mouse carries out three

main functions: pointer movement, menu operations, and nonmenu operations. In general, they correspond to pressing no buttons, pressing the right button, and pressing the left.

Pointer movement. The mouse makes pointer movement fluid and natural. You move the mouse about on a clean, flat surface, and a pointer (arrow, cursor, hand, paintbrush, crosshairs, whatever) reproduces its path on the display. While arrow keys can move the pointer only vertically or horizontally, the mouse can take any course you want. It's the difference between driving through city streets and flying over them.

Menu operations. The mouse controls the menus with the button on the right, the menu button. The menu button is the more limited, less active of the two, dealing essentially with the pull-down menus at the top of the screen (though you can program it to take care of other tasks).

Nonmenu operations. For the remaining mouse operations, you use the lefthand selection button. The selection button is a generalist. Its functions are wide-ranging and diverse. Among other things, it lets you select icons, open them, deselect them, place the cursor, and drag.

Dragging is one of those operations that take a second to learn and forever to describe. To drag, you place the pointer at a particular spot, press the left button, pull the mouse, and release. Among other things, this act will transport items, select areas, and draw lines. To move an icon across the screen, for instance, you just drag it to the new locale and release. Dragging also lets you select an area. In word processing, you can select a whole sentence for deletion by simply dragging the cursor over it. Finally, dragging lets you draw. In graphics programs, you drag a paintbrush across the screen to create a line.

By the way, the left button is not necessarily restricted to nonmenu operations. Like its neighbor, it can control menus, as it does in Electronic Arts's *One-on-One.*

The keyboard. The Amiga keyboard is comfortable and capacious. It has 89 keys, divided into two parts: an alphanumeric keyboard and a numeric keypad. The former has the QWERTY layout of character keys within a rim of command keys. Touch typists will notice that the F and J keys have tiny pimples, for easy finger alignment, and the 5 on the numeric keypad has one as well. In addition, the Amiga CAPS LOCK has an ingratiating red light, which turns on while the key is engaged. The keyboard has all the major command keys,

including ten function keys across the top, a CTRL (Control) key, two ALT (Alternate) keys, two A (Amiga) keys, four arrow keys, and a HELP key. We cannot go over all the keyboard's features, but some of them are worth highlighting.

Reset. Reset is particularly useful on the Amiga because it lets you start the computer over again without reinserting Kickstart, and thus eliminates a great deal of dead time. To reset, you press CTRL and both of the A keys simultaneously. The operation is deliberately ungainly to avoid accidental vaporization of the program in RAM.

Extra character sets. The two ALT, or alternate, keys at either end of the bottom row open up new realms of characters. Press ALT and 6, for instance, and you get the paragraph symbol (¶). Press ALT and other keys and you can generate Greek letters, the angstrom sign (Å), the pound sterling symbol (£), and many more. If you press both ALT and SHIFT along with the character keys, you get a fourth set of symbols.

Mouse tasks. The left and right Amiga keys—shown by filled and hollow *A*'s respectively—are mouse surrogates and carry out the three basic mouse functions:

1. Positioning. Press an Amiga key and an arrow key, and the pointer will scuttle away in the direction of the arrow. To move it faster, press Amiga-SHIFT-arrow.
2. Menu operations. Right Amiga–right ALT duplicates the effect of pressing the right button. You hold both keys down and press the arrow keys until the pointer reaches the menu item you want. When you release, the command executes.
3. Nonmenu operations. Left Amiga–left ALT duplicates the effect of pushing the left button.

Shortcuts. The Amiga keys are awkward and lumbering at many of these tasks—no competition for the agile mouse. But there's one way they can beat the mouse cleanly. They can be shortcuts. For instance, to select a menu item without touching the menu, you press right Amiga plus a predefined key, say, Q. A menu operation like Quit then ensues. The left Amiga plus a character key generates nonmenu commands. Shortcuts can greatly streamline program use.

Fine cursor movement. The Amiga has arrow, BACK SPACE, and DELete keys for moving the cursor a few letters back and forth, a task the mouse does not excel at. The four arrow keys normally move mouse-wise, above text rather than

through it. Hence, the left arrow key does not delete. It lets you easily backtrack a few letters, insert some characters, then return to where you were. To delete, you use BACK SPACE and DEL. BACK SPACE moves the cursor left through text, wiping it out. The DEL key deletes to the right, moving prose toward the cursor.

The keyboard and mouse are the main avenues of communication into the Amiga, and in most cases you can use whichever one you like. But most people use both. Each has special virtues, and together their resonance heightens the impact of the whole machine.

Other input devices. The Amiga, of course, works with the many other input devices available, such as the trackball, joystick, digitizer pad, image processor, optical scanner, microphone, and "piano" keyboard. Some of these are particularly interesting for the Amiga:

Image processors. Image processors translate analog visual images from video cameras into digital ones for display on the screen. Some image processors will also accept input from VCRs. This capacity, called frame grabbing, lets you freeze a frame of, say, Orson Welles in *Touch of Evil*, then feed it into the computer. Once it's there, you can alter it, add sound to it, print it out, and even animate it. An early image processor for the Amiga comes from A-Squared, of Oakland, California. It plugs into the expansion bus and takes input from video cameras, other computers, laser disc players, and VCRs. It stores an image in eight shades of gray at the minimal Amiga resolution of 320 × 200, but the company is also talking about an upgrade to 16 colors at 640 × 400. It was slated to cost between $250 and $300.

Audio input. First-rate audio is relatively new on personal computers, and audio input devices like microphones and "piano" keyboards are becoming available for the Amiga. Cherry Lane offers a 49-key keyboard for $99. You can also play melodies on the alphanumeric keyboard, but it takes a little getting used to.

Memory

The Amiga's multitasking, video, and audio all thrive on memory, and the Amiga has it. It comes with good-sized RAM, which can be expanded easily and dramatically. It also

takes 880K disks and up to four disk drives, and a hard disk is already available for it.

RAM. Technically, the Amiga has 512K of RAM built in, though, since half of it is set aside for Kickstart "ROM," we generally say it has 256K of RAM. That's enough for some programs. But the computer's more impressive feats profit greatly from more, and the Amiga's open architecture makes RAM expansion easy.

Commodore offers a 256K RAM add-on card. It looks rather like a large, metallic harmonica and plugs into the Amiga in the front. You detach the middle third of the front case by squeezing gently on the top and bottom and slowly pulling out. Then you follow the simple directions and plug the extra RAM in. This act instantly doubles your available RAM to 512K. The cartridge costs $195.99.

You can also buy an expansion module called the T-card from Tecmar, of Solon, Ohio. This device snaps onto the expansion bus on the right panel and comes in three sizes—256K ($795), 512K ($895), and 1M ($995). It also offers a clock/calendar with standby battery, a built-in power supply, and ports for further expansion.

You aren't limited to 1M, either. The Amiga's expansion bus will allow up to 8M of RAM, an enormous amount, and though devices to attain this size were not immediately available, they could appear soon. That much RAM will really make the machine bloom.

Disks. The Amiga's internal disk drive accepts 3½-inch double-sided disks which hold 880K. The computer also lets you attach up to three more external disk drives, piggy-back style, each plugging into the drive immediately before it. To run IBM PC software with *The Transformer* program, you may need an external 5¼-inch drive. Currently, the only such drive that will work with the Amiga is made by Commodore and sells for $395, though others may also come out.

Hard disks. Tecmar announced the first hard disk for the Amiga, called the T-disk. The device holds 20M, equal to about 23 Amiga disks, and costs $995. It can fit on the "shoulder" of the Amiga console, so it takes up no additional desk space. Tecmar also offers T-tape, its 20M magnetic tape back-up system, with lights that show track number and tape direction as well as read, write, door, and power status. T-tape can be stacked atop T-disk to save further desk room.

The Processor Chips

The Amiga's internal design might be termed "quasi-parallel processing." It has a CPU—the Motorola 68000—but it also has three extra chips which function on their own. These chips are not CPUs, but rather special-purpose devices, devoted mainly to graphics and sound. They are the Amiga's resident genies.

The Motorola 68000. The 68000 is a powerful CPU. It has 70,000 transistors, slightly more than its name implies. It also has a few variations on the classic CPU layout. For instance, it has three ALUs instead of one. Two manipulate address locations, and the third works on data. In addition, the 68000 has quite a bit of ROM, which substitutes for gates and simplifies manufacture.

The 68000 is a 16/32-bit chip, that is, partway between 16 and 32 bits, but closer to 16. Its registers can hold 32 bits, as can the special register called the counter, and the 68000 can therefore operate on instructions as large as 32 bits. But it can't act on all 32 bits at once. Its ALUs are only 16 bits wide. Thus, it must work on half a 32-bit instruction at a time, and in practice, 32-bit instructions are not often issued. Moreover, its data bus is also 16 bits wide, another major bottleneck. Its address bus is betwixt and between at 24 bits.

The Motorola 68000 has a variety of boons. It offers 18 registers, so the chip can keep plenty of balls in the air at once. It can also access 16M addresses—16,777,216—a vast amount. By comparison, the Apple IIe's 6502 chip can address only 65,536 memory locations.

In addition, the 68000 recognizes 88 assembly language instructions, a large and powerful set. Moreover, the instructions are flexible. The MOVE instruction, for instance, comes in a number of different varieties, and a programmer can use the one that is most convenient. Overall, 68000 instructions are also simple, another virtue.

The Amiga works at 7.16 megahertz (MHz). A speed of 7.16 MHz is pretty fast. By comparison, the 65C02 in the Apple IIc executes at 1.02 MHz, the Intel 8088 in the IBM PC at 4.77 MHz, and the 80287 in the IBM PC AT at 6 MHz.

Output chips. The Amiga is distinguished by three custom chips devoted to graphics and sound, which operate independently from the CPU whenever possible. Designed by Jay Miner, they greatly reduce the waiting line for the CPU

and make the Amiga much faster than other 68000 machines like the Macintosh. Moreover, they are highly specialized and account for the Amiga's remarkable video and audio.

During development, the chips bore the proper Victorian names Agnes, Daphne, and Portia (as well as Agnus, Denise, and Paula and, inevitably, Huey, Dewey, and Louie). Logically, they are a unit, but since their circuitry couldn't fit onto a single chip, they were divided into three:

- Agnes is the animation custom chip. It contains a mix of things: the blitter, which quickly draws lines, fills spaces, and manipulates shapes; the copper, which controls and coordinates the other two chips like a CPU; and a traffic signal regulating the direct access of memory.
- Daphne is the graphics custom chip. It manipulates the display on the screen, taking care of two independent screens at once and coordinating movement of the little sprites.
- Portia not only regulates various ports, but also handles the four sound channels in the Amiga. Officially, it bears the five-footed name of "peripherals/sound custom chip."

The combination of the 68000 and Agnes, Daphne, and Portia is extremely potent and leads to output of ringing quality.

Output Devices

Let's take a brief look at output devices available for the Amiga.

Video. The Amiga can use the four main kinds of video display: television and monochrome, composite color, and RGB monitors.

A television set will work with the Amiga, though, as on other computers, it has certain limitations. A line of text on TV can be only 60 characters wide, including margins, so word processing documents are somewhat narrower. In addition, TV allows 3616 different colors, rather less than the 4096 of the color monitor, but still lavish. TV also restricts you to low resolution, while a monitor lets you use high or low.

The Amiga works with monochrome, composite, and RGB monitors. The first two plug into the Video port, the last into the RGB port. The monochrome and RGB offer 80-column text, and the RGB gives you excellent graphics as well. Commodore offers a fine RGB monitor at a reasonable price. Its

screen is 13 inches across diagonally and fairly looms over the console. You turn it on by pushing an inconspicuous squarish button labeled "Power," at which the thin horizontal light above it goes on. Like the computer, it has a host of vents to dissipate heat, and the monitor dies if you block them. A panel along the bottom pulls down to reveal a string of control knobs, which let you adjust brightness, contrast, and other factors.

Speakers. The Amiga RGB monitor comes with a speaker inside it, on the lefthand side. But you are not confined to the speaker in the Amiga monitor. You can hook the two audio ports up to stereo speakers and expand the sound. You can also plug 1/8-inch headphones into the tiny, white-rimmed opening near the speaker, so if people are trying to work, study, or sleep nearby, you can still make music or play a raucous game.

The Amiga has a voice synthesis capacity built into its operating system, and languages like BASIC let you make the computer speak easily. You can also vary the speed and pitch of the voice and change it from male to female. The Amiga's voice can be made to sound remarkably authentic.

The printer. The Amiga comes with drivers for a number of different printers and allows you to add other drivers as they become available. These include daisywheels, dot-matrixes, lasers, a thermal-transfer, and an ink-jet.

The computer runs five daisywheels: The Alphacom Alphapro 101, the Brother HR-15XL, the Qume LetterPro 20, the Diablo Advantage D25, and the Diablo 630. The first four of these range in price from $400 to $795 and are all rather similar. The last, the Diablo 630, ECS version, is clearly the most substantial. It can print 200 different type styles at 40 characters per second (cps), and each element is capable of 192 different characters. The printer costs $1,995, and is something of a standard in the computer world.

Drivers for dot-matrix printers include one for the Commodore MPS 1000 (not yet available), one for the Epson JX-80, and a generic Epson driver for that company's FX and RX series printers (and the many other printers compatible with those models). The JX-80 is distinctive in that it offers color. It has nine pins, prints seven colors (black, red, orange, yellow, green, blue, and violet), and sells for about $599.

Okidata's color Okimate 20 is the thermal-transfer that works with the Amiga. It boasts a very low cost, around $268, though to use it with the Amiga, you must also buy a special cartridge. Its printhead has 24 tiny heat elements, so its quality is outstanding. It gives you black and seven colors, and about 100 shades. It prints any kind of text or graphics, at either 80 cps or 40 cps. However, like other thermal-transfers, it eats up ribbons very fast. A black-and-white ribbon used for text may last for 75 pages, but a color ribbon used for graphics is useless after 8 to 15 pages.

The color Diablo C-150 is a very quiet, trim ink-jet printer weighing about 24 pounds and costing about $1,250. It yields brilliant quality, but it will probably mainly be used for graphics. It prints text at a painful 20 cps, as slow as a daisywheel. The printer also requires special paper, and setting it up can be an intricate business, though documentation is very clear.

Laser printers supported include Hewlett-Packard's LaserJet and LaserJet Plus. Though the manual doesn't mention it, Apple's LaserWriter will also run at once on the Amiga, since it was designed to be fully compatible with the Diablo 630 daisywheel. This famed machine prints eight pages a minute, yields resolution of 300 dots per inch, and has 512K of ROM and 1.5M of RAM, far more than most personal computers. It costs $5,999.

Printer control. The Amiga not only comes with many different drivers, but also lets you specify how the printer will work. It does so with the second and third screens of Preferences.

The second screen, the Printer Requester, takes care of the basics. You use it to indicate which printer you are using and the port it's plugged into, as well as such properties of printout as page size, margin size, number of characters per inch (pitch), and number of lines per inch (spacing). In addition, you can set print quality (draft or letter) and paper type (fanfold or single).

Once past the Printer Requester, you can move onto the third screen: the Printer Graphics Requester. Here you can play with the image before it reaches the printer. Shade, for instance, lets you choose among printing in color; in grayscale, which renders colors as shades of gray; and in black-and-white, which prints colors as black or white according to their brightness level. You set that line of demarcation with

the Threshold scale across the top of the screen. In addition, the Image option lets you reverse the picture like a photographic negative, and Aspect allows you to print sideways.

Emulation

Prior to the Amiga's introduction, it was common knowledge that it would emulate the IBM PC by hardware. The device even had a name: the Trump Card. At the unveiling, however, spectators at Lincoln Center saw it emulate with a software program, *The Transformer.* There was general amazement.

To use the $99 *Transformer,* you insert the 3½-inch disk into the Amiga and soon see the MS-DOS screen, which presents various menu offerings. You then insert either another 3½-inch disk into the Amiga's internal drive, or a 5¼-inch disk into an external drive. At this point, the Amiga essentially loses its identity. It becomes an IBM PC. In exchange for running PC software, you forfeit the Amiga's graphics, audio, multitasking, and other bounties.

The Transformer does not yield a 100 percent complete IBM PC, at least not yet. Its first version was incompatible with programs requiring the IBM graphics upgrade card, though its second version, due by early 1986, was to fix this problem. In general, Commodore has not specified how compatible *The Transformer* will be, beyond saying that it will work with the 25 bestselling PC programs. That's a lot, and for most people compatibility should not be a serious concern.

Currently, *The Transformer* seems to run IBM software at around 60 percent of speed. Its disk access is about the same as the IBM's, but certain other features like graphics are about half as fast. The lag isn't appreciable with word processing, since this application doesn't depend on high velocity. On the other hand, large spreadsheets with a great deal of calculation will cause the emulating Amiga to bog down a bit. To minimize this problem, Commodore has announced a $100 hardware accelerator. This slender device fits onto the expansion bus and consists of extra RAM and a Program Assistance Logic (PAL) chip, which together boost the performance of *The Transformer.*

Video Powers

The Amiga can display 4096 colors at up to 640 × 400 and has powerful built-in animation features as well.

Colors. The Amiga achieves 4096 colors because it can di-

rect the cathode in the CRT to shine its electron beam at 16 different intensities. Each intensity causes a phosphor pixel to glow in a different shade. Since the beam strikes red, green, and blue pixels, which fuse to make the onscreen colors we see, the Amiga can render 16^3 colors, or 4096.

The computer stores code for these colors in 32 color registers which are 12 bits wide. At this size, each can hold 4096 numbers and hence denote any hue in the color pool. The contents of each pixel's address in RAM refer to one of these registers, which in turn refers to the color. This approach is called color indirection, and it saves a great deal of internal memory.

There's an obvious price for it. Color indirection limits the Amiga to 32 colors onscreen at any one time. We can load any color we want into a particular register and so pick and choose among the 4096, but we cannot get more than 32 at once. Yet 32 is more than most computers allow and does not really hinder enjoyment of the machine. Moreover, color indirection is not just a space saver. It is also a performer.

For instance, if you change the hue in one register, you instantly change it everywhere it appears on the screen. It cannot be otherwise, since every pixel that refers to that register must take on the new color. It's a significant power and makes for lightning color changes.

Color indirection also makes it easy to draw single lines in multicolored segments. You arrange for the paintbrush to paint in the color of one register for, say, half a second, then in another for half a second, and so on, so that as you pull it across the screen, it leaves a trail of many tints. In *Graphicraft,* this technique is called Cycle Draw.

Moreover, you can make onscreen colors shiver with iridescence. You arrange to move the contents of register 1 into register 2, and 2 into 3, and so on, like musical chairs, and the colors on the screen will cycle rapidly. The effect can be dazzling. You can alter the colors of concentric circles so they seem to be expanding, or, if you have painted a line with Cycle Draw, you can make the color segments appear to travel rapidly down the line. And if you get the entire screen flashing, it looks like a light show.

Resolution. The Amiga has two main levels of display, low and high, which differ principally in resolution and number

of onscreen colors. Each type has two subsets of resolution/color capacity: normal and interlaced.

In low resolution, the normal mode is 320 pixels wide × 200 high. If you move up a notch to the interlaced mode, the computer will spray the screen with twice as many lines and give a picture 320 pixels wide × 400 high. Normal requires 40K of memory, interlaced 80K. In both modes, the palette can hold 32 colors.

At high resolution, the Amiga grows resplendent. High resolution also has two levels, normal and interlaced, and both allow a palette of 16 colors. Normal is 640 pixels wide × 200 high, and interlaced, 640 × 400, the finest resolution the Amiga has, and among the finest of any personal computer. Normal requires 64K and interlaced, 128K.

The Amiga's two interlaced modes work somewhat like interlaced fingers. In the first 1/60 second, the CRT electron gun covers a 320 × 200 or 640 × 200 screen, leaving empty spaces between each line it strikes. In the second 1/60, it covers another 320 × 200 or 640 × 200, but it shifts slightly down to fill in the empty lines. It's an easy task for the Amiga's video chip. The phosphor glow from the first display lingers on, and the mind knits the two images into one.

That's the theory, anyway. In practice, the phosphors from the first spraying have started to fade by the time the second one arrives. Hence, the two images don't quite merge. The result is quiver, slight but noticeable. There's no way to avoid it short of reconstructing the monitor so that it shows images faster than 60 times a second. The Macintosh uses this approach, but of course it displays only in black and white.

There is actually a third type of resolution, called hold and modify. Like low resolution, it comes in either 320 × 200 or 320 × 400. However, it lets you put the entire 4096 colors in your palette at once. Hold and modify works on a completely different basis from color indirection. Basically, it is a relative rather than absolute system. It defines each pixel in terms of the pixel just before it. Hence, it holds the previous value long enough to modify it and get the new value. It seems poorly suited for animation and other shifting images and will thus probably be used mainly for static pictures.

Playfields. The Amiga's screen is more than just the product of its colors and resolution levels. Its graphics chips

give it special powers. They confer a structure on the screen and grace it with brilliant prowess in animation.

The first and most obvious element of that structure is the playfield. A playfield is essentially an independent screen, the same width as the screen itself, but of variable height. Two playfields are available on the CRT, and each can have eight different colors.

Playfields have interesting properties. First, one playfield can have priority over the other so that it lays over it. At the same time, parts of the dominant playfield can be transparent, so you can look through and see what's happening on the playfield below. This characteristic fits playfields well for games. In *Skyfox,* for instance, one playfield, the cockpit, can have transparent spaces through which you view the second playfield, the hostile world around. Both work together to heighten the effect of soaring over countryside.

Sprites. Sprites originated as a hardware solution to the difficulties of animation. They are small objects that move across the playfields. A sprite can be 16 pixels wide, that is, 1/20 of the screen in low resolution. It can also be as tall as the screen. The Amiga offers you eight of them, and you can get more by reusing some on the same screen. Each pixel of a sprite can have one of four colors. It's also possible to attach two sprites to each other, making one sprite with the capacities of two, and hence with a range of 16 different colors.

Sprites have several features in common with playfields. First, you can give them a hierarchy so that one will always appear atop another. Indeed, you can have up to seven layers of priority. In addition, you can make one of their colors transparency, so you can see through one sprite onto another. In fact, in some ways you can think of the playfield as simply a large sprite and vice versa. They have different hardware backgrounds, but they can work in very similar ways.

Animation. The Amiga's talent for animation really brings it alive. A computer screen can glitter like a handful of gems, but it's still static. Motion gives it past, present, and future, as well as verve and élan, and it can bewitch us.

The Amiga has two animation systems, one for playfields and one for sprites.

The blitter—part of the Agnes chip—controls playfield animation and confers noble capacities upon it. It works at

high speed, always a blessing for animation, and transfers images from one place to another. Such an operation means moving code about in the bitmap, and *blitter* is a telescoping of "block image transferrer."

Animation on the playfield works like this. A programmer indicates an image on a background. The image and background are saved in memory. The programmer can then tell the blitter to move the image around as a block. With playfield animation, you can shift several dozen objects as well as fill spaces quickly and draw lines at an eye-popping one million pixels per second.

The second kind of animation is sprite animation. It works faster than playfield animation and generally controls the darting about of sprites. Intriguingly, if you run out of sprites, you can always use the blitter to set up other independent, spritelike objects, of which there is no limit. Playfield animation is so good and can replicate sprite animation in so many ways that the latter has lost some of its importance.

Both types have a built-in collision detection capacity. The Amiga can tell when two sprites, a sprite and a playfield, or two playfields have bumped into each other. It's a useful feature. In games, objects strike each other all the time, and if the hardware can sense the impact, the software can move on to better things, making the game richer and faster. Collision detection also lets you confine roving objects to a prescribed territory.

Aegis Animator, from Aegis Development of Santa Monica, California, was one of the first animation programs for the Amiga, and it shows what such software can do. It rests on the concept of the *tween,* an automatic, mobile transition from one figure to its successor. For instance, if you move a polygon from the left side of the screen to the right, the tween will play back the shift from start to finish. It's like a tiny movie.

Tweens on the *Aegis Animator* are not limited to simple shifts. They can move objects on complex courses, rotate them around three different axes, expand or shrink them, change their shapes and colors, and move them in front of or behind other objects. A single tween can do all these things, and you can link tweens together to form a longer piece of animation.

The software has further capacities. It permits control over global features, such as speed of playback. It also has a storybook mode. Storybook divides the screen into nine equal

compartments, where you can cut and paste objects from one animation into another or splice whole animations together.

The genlock interface. The genlock interface is a means of working with external video signals, like those from a VCR, video camera, or even another computer. With a genlock interface, you can read in a video frame, like, say, a picture of Molokai, and use it as a background for graphics on your Amiga. It's a powerful way to manipulate images.

Audio Powers

The Amiga also offers splendid audio, which can enhance almost everything the computer does. Not only does the Amiga yield sound of very high quality, but it lets you manipulate it in a remarkable number of ways.

The Amiga can act as a digital music synthesizer. That is, it can form notes out of their basic constituents by synthesizing them. Sound is simply waves, and the computer can describe the waveform of any note by assigning numbers to it over time. It's like a connect-the-dots puzzle, where each number specifies a dot's position. Link them together and you get a wave with a particular frequency (pitch), height (volume), and shape (timbre). You can then feed the signal into a digital-to-analog converter. The converter turns it into electrical waves, which flow to the speaker and emerge as sound.

The Amiga affords great control over sound. It lets you set the volume envelope, that is, the loudness trajectory—attack, decay, sustain, and release—of individual notes. It also gives you 64 different levels of volume control, a very wide range. You can also regulate timbre, the quality that distinguishes a B-flat on an oboe from the same note on, say, a trumpet. In addition, the Amiga imitates instruments like the vibes with a realism that is truly startling. Such mimicry is not just a trick. Retail prices for a set of vibes start at $2,500.

The computer has four independent channels of sound, and their autonomy is significant. Since you can program each channel separately, each is technically a synthesizer, and the Amiga is really a quartet of the devices. You can also combine channels in pairs to achieve bona fide stereo so that hooking up two speakers to the audio ports yields more than just extra volume. And the Amiga does not limit you to four different sounds. Each channel itself can play multilevel tones, so the computer can emit a panoply of sounds at once.

You can also use any of these channels as a music sequencer. A sequencer is an electronic instrument that generates a series of notes over and over again. It's not much use on its own, but it has an important role in creating musical background, especially in rock, which thrives on a sense of throb beneath the surface. Groups like Tangerine Dream and Kraftwerk have made extensive use of sequencers.

Moreover, with the proper interface the Amiga can do sound sampling. It's a marvelous attribute. An audio digitizer allows you to attach a microphone to the Amiga and play in a specific sound, for example, of a finger snapping. The Amiga digitizes the waveform of that sound and stores it. Now it's simply one more instrument to the computer, just like the vibes. You can play out melodies with it, alter its volume envelope, send it through filters, and manipulate it generally. And, of course, you aren't limited to finger snaps. You can read in the sound of musical instruments, your own voice, a cat's meow, a washing machine, the Amiga disk drive, anything you want. With several such sounds, you can create a polyphony of the parlor.

The Amiga also has a built-in faculty for voice synthesis, male or female, in a range of eight to nine octaves. Its base-level quality was outstanding for personal computers, though few people would confuse it with human speech. It tended to partition diphthongs like *oi* into *oh* and *ee,* and to articulate those unstressed vowels—like the second *i* in *imitate*—that we in English reduce to a schwa. But programmers are working with the Amiga on a phonemic level, and it is already addressing us in very lifelike tones indeed.

Software. At press time, several interesting programs were in the offing, and they give an idea of the Amiga's audio potential.

Musicraft, from Cherry Lane, is a basic synthesizer program. It gives access to the four sound channels of the Amiga and lets you control volume, timbre, and the other fundamental elements of sound. It lets you play on your Amiga keyboard, or, if you want greater ease and comfort, on a piano-style keyboard.

Harmony, also from Cherry Lane, is an accompaniment program. It offers a choice of songs, initially from the Beatles and Lionel Richie. Each has five parts. You sing or play one part of the tune, and the Amiga generates the four-part accompaniment. If you speed up, the computer speeds up. If

you play softly, the computer plays softly. You don't have to hit each note exactly, since the software deduces your place if you get close enough. The company planned to sell this product for $79.

Texture, another Cherry Lane program, will let you modify a prior digital recording. You can have eight different tracks and manipulate any note on any track. For instance, you could filter out pitches in a range you don't want, shift the key of what you've recorded, or alter the tempo. It uses graphics to show you the recorded notes and aid modification. It was to be priced at $199.

All of this Cherry Lane software is designed to work together. In addition, the company intended to release a sound sampling program as well as educational music programs, such as one to train the ear. It may even offer a voice-library manager, a database for sounds created by sound sampling.

Cherry Lane will not be the only company selling Amiga music software. Electronic Courseware Systems plans a line of educational music software to teach, among other things, blues, keyboard chords, intervals, jazz, and piano sight reading. Passport Designs, of Half Moon Bay, California, will market them as well as issue its own music synthesizer program, *The Music Shop.*

The MIDI interface. Finally, the Amiga is compatible with MIDI, the Musical Instrument Digital Interface. MIDI is a recently standardized means of communication between synthesizers. It allows you to hook up several of them to your Amiga and control them all from there.

For instance, if you use MIDI to attach two digital or analog synthesizers to the Amiga, you can play the computer, and both "slave" synthesizers will respond to your commands. You will be playing three machines at once, and since each may specialize in different timbres, you gain great range. You can also, if you like, store prerecorded music in the adjunct devices so that it accompanies you. You can thus achieve in live performance the kind of layered sound you otherwise get only in a recording studio with multiple tracks. In fact, you can attach up to 16 other devices to the Amiga, including not just synthesizers but digital drums, which will control the timekeeping just as a drummer does in a live band.

Moreover, you can create other, even more wide-ranging effects. J. L. Cooper Electronics, of Marina del Rey, California, sells a MIDI Lighting Controller, which lets you prerecord lightning and special effects to synchronize with the music. MIDI can even allow a good composer to dispense with the orchestra and record an entire motion picture sound track at home.

The MIDI interface is a hardware device and does not come with the Amiga. You have to buy one, for about $60 to $90. But for music professionals, it's an open sesame.

The Amiga itself is an open sesame for programmers. It unlocks an alluring new world, where software can glisten and sing as never before.

Chapter 2

BASIC Programming

C. Regena

BASIC Programming

C.Regena

Writing a program is a way to get the computer to do what you want it to do. Amiga BASIC by Microsoft is the version of BASIC that comes with the Amiga. It is a powerful and versatile language that allows the BASIC programmer to use most of the features built into the Amiga.

If you have programmed in BASIC before, you will find that Amiga BASIC is very similar to other versions, with additional commands for some of the special Amiga features. Graphics and sound have their own commands that add to the capabilities of Amiga BASIC.

A program is a set of instructions that tell the computer to execute a procedure in a certain order. To use the program, you will need to run it. There are three ways to tell your Amiga to run a BASIC program which is in memory: Either click in the Output window, then type RUN and press RETURN; press the right Amiga key and the R key; or use the mouse to go to the Run Menu and select Start.

To write your own program, you need to have the cursor in the List window. If you type a statement in the Output window, it will be executed immediately. If you type a statement in the List window, it becomes part of the program. If the List window is not visible, you must either type LIST and press RETURN, press the Amiga key and the L key, or use the mouse to go to the Windows menu and select Show List. To move the cursor from one window to the other, position the mouse arrow in the desired window and click the left mouse button once. On programs that require user interaction, the cursor will need to be in the Output window (click the arrow there even though you may not be able to see the cursor).

Unlike many other versions of BASIC, Amiga BASIC does not require line numbers. If a line does need to be referenced, it may have either a line number or a line label. To use a line label, type a word and then a colon. To reference the line, do not type the colon.

WHEEL:

When you need to reference the line, use

GOSUB WHEEL

A line number must be an integer from 0 through 65529 and must start in the leftmost column of the line. A line label must start with an alphabetic character, but it may contain any combination of letters, numbers, and periods (other than reserved words). It may be up to 40 characters long and must end with a colon. A program can have both a line number and a label.

With computers that use line numbers on every statement, it doesn't matter what order you type lines in—the computer will rearrange the lines in numeric order to list or run the program. In Amiga BASIC, the program will stay exactly as you type it in the List window. Spaces and blank lines are *not* suppressed. Line numbers do not need to be in ascending order. If you need to add a line, you can move the cursor up to the proper place, press RETURN to get an extra line, then type the new line. The physical order of the lines is important.

Variables

Amiga BASIC allows several types of variables. String variable names end with the dollar sign (such as ADDRESS$) and may include letters, numbers, and symbols. Strings may be up to 255 characters long.

Integer variable names end with the percentage sign (such as SCORE%) and indicate a whole number.

Numeric variables may be either single-precision or double-precision. The single-precision number may be seven or fewer digits, be in exponential form denoted by E, or end with an exclamation point (such as 123! or N!). The double-precision number has eight or more significant digits, is designated by D in exponential form, or ends with a pound sign (such as 123.45# or X#). You may prefer to designate variable names at the beginning of a program with DEFSTR, DEFINT, DEFSNG, and DEFDBL.

Numeric functions are usually three-character abbreviations followed by a numeric expression in parentheses. The expression may be a constant, a variable, an arithmetic expression, or another function. Although the examples given here are functions used in simple PRINT statements, the functions can be assigned to variables or can be combined in other numeric expressions including other functions. All functions involving angles use the radians rather than degrees.

The remainder of this chapter is divided into three parts. The first and largest is a dictionary of Amiga BASIC words with explanations of how they are used and sample statements or programs illustrating their use. Following the dictionary of words are sections on file processing commands and subprograms.

In all the function descriptions in this chapter *n* represents a numeric expression and *s* represents a string expression.

Optional parameters are shown in brackets; *x* and *y* indicate numeric expressions for coordinates in graphics. If a program is given, it will start with REM and end with the END statement.

ABS(n)

The ABS(*n*) function returns the absolute value of the numeric expression *n*. If the number is positive or zero, the value returned is equal to that number. If the number is negative, the value returned is equal to the positive value of the number.

```
PRINT ABS(28)
PRINT ABS(0)
PRINT ABS(-5.2)
```

AND

AND is used in IF-THEN statements to combine relational expressions for a conditional branch. AND indicates that both relations must be true for the condition listed after the word *THEN* to be executed. AND may also be used to show a true or false condition as numeric results: −1 is true; 0 is false.

```
IF SC=10 AND P$="RED" THEN PRINT "RED WINS!"
C=(A=0) AND (B=0):PRINT C
```

AREA and AREAFILL

AREA *(x,y)* or AREA STEP *(x,y)*
AREAFILL *[m]* where *m* is 0 or 1

AREA specifies points in graphics to be joined in a polygon, then AREAFILL joins those points and fills in the polygon with the default solid color or a specified pattern (see PATTERN). AREA statements may use actual numbers or variables specifying the coordinates or may use the STEP option which gives relative distances. AREAFILL 0 is the default value and fills the area with the area pattern. If pattern has not been

specified with the PATTERN statement, the fill is solid. AREAFILL 1 inverts the fill pattern.

```
REM AREA
AREA (10,20)
AREA (50,70)
AREA (25,90)
AREA (10,20)
AREAFILL
X=50:Y=20
AREA (X,Y)
AREA STEP (10,20)
AREA STEP (-10,20)
AREA STEP (-10,-20)
AREA STEP (10,-20)
AREAFILL 1
END
```

ASC(*s*)

ASC(*s*) returns the ASCII code for the first character of the string *s*. ASCII is the American Standard Code for Information Interchange. (See Appendix A.)

```
REM ASC
PRINT "KEY PRESSED","ASCII"
PRINT
AGAIN:
  K$=" "
  WHILE K$=" ":K$=INKEY$:WEND
  PRINT K$,ASC(K$)
  GOTO AGAIN
END
```

ATN(*n*)

ATN(*n*) returns the arctangent of the numeric expression *n*. Arctangent *n* means the angle whose tangent is *n* and will be expressed in radians.

```
PI=4*ATN(1):PRINT PI
D=ATN(R)*(180/(4*ATN(1)))
```

BASE

See OPTION BASE.

BEEP

BEEP is a simple command that will make a short sound and blink the screen.

```
REM BEEP
BEEP:INPUT "ENTER A NUMBER",N
BEEP: PRINT N
END
```

BREAK ON, BREAK OFF, BREAK STOP

BREAK ON will activate ON BREAK error trapping. BREAK OFF ends ON BREAK error trapping. BREAK STOP suspends ON BREAK error trapping until the next BREAK ON instruction.

```
REM BREAK
BEGIN:
  BREAK ON
  ON BREAK GOSUB YOUSTOPPEDIT
UNTILSTOPPED:
  PRINT "PRESS AMIGA AND PERIOD KEY"
  GOTO UNTILSTOPPED
YOUSTOPPEDIT:
  FOR A= 1 to 20
  PRINT "THANKS"
  NEXT
  BREAK OFF
  RETURN
END
```

See also ON BREAK.

CALL

CALL is described in a separate section entitled "Sub-programs," following the dictionary of Amiga words.

CDBL(*n*)

CDBL converts a number to double-precision so that it can be used in calculations with other double-precision numbers. Keep in mind that the accuracy will still be just to the place the original number was.

```
REM CDBL
X=SQR(12)
PRINT X,CDBL(X)
END
```

CHAIN

CHAIN is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

CHR$ (n)

CHR$(*n*) returns the string character corresponding to the ASCII code number *n*. (See Appendix A.)

```
REM CHR$
FOR C=50 TO 70
  PRINT C,CHR$(C)
NEXT C
END
```

CINT (n)

CINT(*n*) converts the number *n* to an integer by rounding. The INT function does not round, but gives the closest whole number smaller. The FIX function truncates the decimal portion and returns the whole number portion.

```
PRINT CINT(3.6),CINT(3.2)
PRINT CINT(-3.6),CINT(-3.2)
```

CIRCLE

CIRCLE [STEP](*x,y*),*r*[,*c*][,*s*,*e*][,*a*]

The CIRCLE command draws a circle with the center point at the coordinates (*x,y*) and a radius *r*. You may also specify a color number *c*. The next two parameters are a start and end point for drawing arcs. The last parameter is the aspect, or the height/width, ratio which will enable you to draw an ellipse; *c*, *s*, *e*, and *a* are optional.

To use the start and end options, imagine a round clock face. A start of zero would be at three o'clock, and the angle goes counterclockwise. The angle is expressed in radians. For example, starting at zero and using π radians, or 3.14159, as the ending point will draw a half circle.

If you include the STEP option, the x and y coordinates will be relative to the most recent pen position. For instance, if the pen position is (5,5), CIRCLE STEP(30,10) will place the pen at (35,15).

```
REM CIRCLE
CIRCLE (50,30),30
CIRCLE (50,60),20,3
CIRCLE (50,100),20,1,0,2
CIRCLE (90,60),25,1,,,3
CIRCLE (140,60),25,2,,,.2
END
```

CLEAR

CLEAR sets all numeric variables to zero and string variables to null.

```
REM CLEAR
X=5:PRINT X
CLEAR
PRINT X
END
```

CLEAR can also specify an amount of memory to be allocated to the Amiga BASIC data area and to the system stack. This is used to change default values, for example, when you want to use less of the default space allocated for graphics and more for numeric data.

CLEAR, *BASICdata, stack*

The *BASICdata* numeric expression must be 1024 bytes or greater. The *stack* expression also must be 1024 bytes or greater. If you leave out the data allocation, but use the stack, the commas must indicate the places.

```
CLEAR, 25000 + 40960
CLEAR, 40000,2000
CLEAR, 6000
```

CLNG *(n)*

CLNG*(n)* converts a number *n* to a LoNG integer by rounding. Ordinarily when a number over a million is printed, it is printed in exponential form—a number times ten to a power, which is in E format on this computer. If you prefer to keep all the significant figures and write the number as we are used to seeing it in decimal form, you can use a LNG number. Notice, too, that if you have a fraction, this function rounds rather than truncates the decimal portion.

```
REM CLNG
FOR T=1 TO 6
  READ X(T)
  PRINT X(T),CLNG(X(T))
NEXT T
DATA 15032312,222.345,254445887.78
DATA -22777654.67,-3,-22997636.32
END
```

CLOSE

CLOSE is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

CLS

CLS CLearS the active Output window (erases everything) and starts the cursor back at the top left corner of the Output window. Although when you run a program in Amiga BASIC the screen automatically clears at the beginning, you may want to use CLS later in the program to start with a new clean screen.

```
REM CLS
FOR X=1 TO 20
  PRINT TAB(X);X
NEXT X
FOR D=1 TO 2000:NEXT D
CLS
END
```

COLLISION

COLLISION may be either a function or a part of another statement.

COLLISION(−1) returns the number of the window where collision identified by COLLISION(0) has occurred.

COLLISION(0) returns the number of an object that collides with another object.

COLLISION(*i*), where *i* is the object ID number, returns the number of an object that collided with object *i*. If the value returned is negative, the collision was with a window border. The top border is indicated by −1, the left border by −2, the bottom border by −3, and the right border by −4.

ON COLLISION GOSUB *L*, where *L* is a line number or label, goes to the subroutine when a collision occurs.

COLLISION ON turns on event trapping declared by the ON COLLISION GOSUB statement.

COLLISION OFF ends event trapping.

COLLISION STOP prevents execution initiated by the ON COLLISION–GOSUB statement until COLLISION ON is executed again.

See OBJECT for a sample program using COLLISION.

COLOR

COLOR *f,b* defines a foreground color and background color to be used in text printing, drawing, and filling areas. In the default screen, you may use colors 0, 1, 2, and 3, where 0 is blue, 1 is white, 2 is black, and 3 is orange. These colors will be different if you changed them with Preferences. The SCREEN statement lets you change the number of colors up to 32.

```
REM COLOR
PRINT "HELLO"
COLOR 2,1:PRINT "BLACK"
COLOR 3,1:PRINT "COLOR 3 ON 1"
COLOR 2,3:PRINT "BLACK ON ORANGE"
COLOR 0,1:PRINT "I LIKE THIS"
COLOR 1,0
END
```

CONT

CONT continues a program after you have stopped it by either pressing CTRL-C or the Amiga and period keys or used the mouse to Stop the program from the Run menu. To try this command, run this short program, then stop it. Put the cursor in the Output window and type CONT and press RETURN. You can also continue the program by using the mouse and selecting Continue from the Run menu.

```
10 REM CONT
20 X=X+4:PRINT X
30 GOTO 20
40 END
```

COS(*n*)

COS(*n*) returns the cosine of an angle *n* where *n* is expressed in radians.

```
REM COS
LINE (0,100)-(640,100)
FOR X=0 TO 32 STEP .1
  Y=100-(40*COS(X))
  LINE (X*20,Y)-(X*20,100)
NEXT X
END
```

CSNG *(n)*

CSNG*(n)* is a numeric function that converts the numeric expression *n* to a single-precision number for calculations with other single-precision numbers.

```
N#=500000.3572
PRINT N#,CSNG(N#)
```

CSRLIN

CSRLIN is a function that returns the current row position of the cursor, or the line the cursor is on.

```
REM CSRLIN
RANDOMIZE TIMER
FOR J=1 TO INT(10*RND)+1
  PRINT TAB(J);"HI";
NEXT J
PRINT:PRINT "CURSOR IS ON ROW";CSRLIN
END
```

See also POS.

CVI, CVL, CVS, CVD

These words are described in a separate section entitled "File Processing," following the dictionary of Amiga words.

DATA

DATA statements store data in a program. The data list may be numbers or strings, and each item must be separated by commas. If a string contains leading or trailing spaces, or embedded commas, it must be in quotation marks.

A DATA statement may be placed anywhere in the program. As the program is run, DATA statements are ignored until a READ statement is executed, which assigns the data to variables. The first READ statement starts to read items at the beginning of the first DATA statement. There must be enough DATA to satisfy the READ statement or you will get an error message. The DATA type, either string or numeric, must correspond to the variable names in the READ statement. Data items are read in order by the READ statement unless a RESTORE statement is used.

The computer always reads the data in order, no matter how the DATA statements are typed. The computer keeps track of a pointer so it knows which is the next data item to be

read. If the READ statement does not use all the DATA items, the extra ones are ignored.

```
REM DATA
FOR T=1 TO 10
  READ AGE,N$
  PRINT N$,AGE
NEXT T
DATA 16,CHERY,14,RICHARD,11,CINDY
DATA 9,BOB,5,RANDY,.2,BRETT,11,ED
DATA 9,BILL,7,JOHN,3,JIMMY
END
```

See also READ and RESTORE.

DATE$

DATE$ returns the date in the computer in the format *xx-yy-zzzz* (month-day-year). You may change the date by selecting Preferences from the Workbench and changing the printed date.

```
V$=DATE$
PRINT DATE$
```

DEF

DEF FN defines a function to use later in the program. It can save typing if you have to use it several places in the program. A function must be defined before it is used. A function can call another function.

```
REM DEF
DEF FNR(N)=INT(N*RND)+1
PRINT FNR(8)
D=6:PRINT FNR(D)
END
```

DEFDBL

DEFDBL defines the specified variables to be double-precision numeric variables, so you do not need the # sign at the end of the variable name. You may specify either the complete variable name or the first letter or a range of first letters. If you specify only one letter (or range of letters), any variable name starting with that letter will be defined.

```
DEFDBL N,LENGTH,WIDTH
```

DEFINT

DEFINT defines the specified variables to be integers and will round the integers. (The INT function does not round.) If you do not define numeric variables, the default is single-precision floating point (noninteger).

```
REM DEFINT
DEFINT I-M
I=2.45:J=3.2:KKK=3.75
PRINT I,J,KKK
END
```

DEFLNG

DEFLNG defines the specified variables to be LoNG integers and will round the integers. Ordinarily, after seven digits, a number is written in exponential format, or a number times E*n* indicating the number times 10 to the *n* power. DEFLNG will keep the numbers in normal written form, but allows for more digits.

```
DEFLNG X,Y,Z
```

See also CLNG.

DEFSNG

DEFSNG defines the specified variables to be single-precision numeric variables, so you do not need to use the ! sign in variable names.

```
DEFSNG AVERAGE,SCORE
```

DEFSTR

DEFSTR defines the specified variables to be string variables, so you do not need to use the dollar sign in variable names. You may specify either the actual variable name or the beginning letter or range of letters.

```
DEFSTR B-D
DEFSTR PERSON,PHONE
```

DELETE

DELETE is used as you're programming or editing to delete or erase program lines. You may specify either a line number or a range of line numbers.

```
DELETE 60        Deletes line 60
DELETE 300-500   Deletes lines 300 through 500
```

DELETE -100 Deletes all lines through line 100
DELETE 700- Deletes lines from line 700 to end

DIM

DIMensions arrays of numbers. If you use a variable with a subscript, the computer automatically reserves 11 elements (numbers 0–10). If you want to conserve memory for fewer elements or if you need to reserve memory for more elements, you do so with the DIMension statement.

```
DIM D(6),A(30)
DIM N$(3,6,4)
```

ELSE

ELSE is used in an IF-THEN statement to give a command if the condition is not true. ELSE may be followed by a line number, label, command, or another IF statement. The IF-THEN statement does not have to include ELSE.

```
10 REM ELSE
20 PRINT "PRESS A LETTER"
30 A$=INKEY$:IF A$="" THEN 30
40 PRINT A$;" - ";
50 A=ASC(A$)
60 IF A<65 THEN PRINT "NO":GOTO 30
70 IF A>123 THEN PRINT "NO":GOTO 30
80 IF A>90 AND A<97 THEN PRINT "NO" ELSE PRINT "YES"
90 GOTO 30
100 END
```

See also IF-THEN.

END

END stops execution of the program. If the cursor is in the Output window, the message "Ok" appears. If the cursor is in the List window, the List window reappears. The END statement is actually optional because the computer will stop execution when there are no more statements to execute. You may also wish to use END to prevent the computer from going to later lines such as subroutines. END differs from STOP. The program can be continued after STOP, but not after END.

EOF

EOF is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

EQV

EQV (equivalence) is used as a logical operator in conditional statements. A EQV B returns true if both A and B conditions are true or if both A and B are false. If one condition is true and one is false, EQV will return false.

```
REM EQV
A=14:B=7:C=2
IF A=B*C EQV A>0 THEN PRINT "1 TRUE"
IF A>B EQV C>B THEN PRINT "2 TRUE"
IF B>A EQV C>A THEN PRINT "3 TRUE"
END
```

See also IF-THEN.

ERASE

ERASE *a* where *a* is an array name allows you to erase arrays within a program. One use of this statement is to conserve memory as a program is running. After an array has been used, ERASE can be used to erase the need for that memory. ERASE may also be used if you want to redimension arrays. ERASE N$ indicates that you no longer need the N$ array.

```
REM ERASE
DIM A(15)
WIDTH 20
FOR C=1 TO 15
  A(C)=C:PRINT A(C);
NEXT C:PRINT
ERASE A:DIM A(18)
FOR C=1 TO 18
  PRINT A(C);
NEXT C
END
```

ERL, ERR, and ERROR

ON ERROR GOTO *l*, where *l* is a line number or label, is a way to do your own error trapping. The line number is the first line of the error-handling subroutine. Once an error occurs, the subroutine is executed. You need to use RESUME to exit from the error-handling routine.

ERROR *n* is a way to simulate the occurrence of a BASIC error, or you can define your own error codes; *n* is an error code number. If you define your own, you should use a code that is not already assigned (check the appendix of your manual for a list of the error codes). If you have a statement ERROR *n*,

the program will stop with the error message corresponding to the code number *n*.

ERL contains the line number where the error occurred. ERR contains the error code for the last error. The following program illustrates these BASIC words. ON ERROR GOTO 50 says that if an error occurs go to line 50, which is the beginning of the error-handling subroutine. Line 10 asks you to input a number. We are setting up an error condition in line 15. If the number entered is greater than 1000, then ERROR 220, which says the error code is 220. Line 35 also tests for an error condition. Line 50 says that if the error code is 220, then print the error message that we defined. The next line says if the error occurred in line 35, then RESUME 30, which means to go back to line 30 and continue the program. If the error did not occur in line 35, RESUME 10 sends the computer back to line 10.

```
REM ERROR
ON ERROR GOTO 50
10 INPUT "ENTER A NUMBER ",N
15 IF N>1000 THEN ERROR 220
20 PRINT "OK"
30 INPUT "NOW YOUR NAME ",N$
35 IF LEN(N$)>8 THEN ERROR 220
50 IF ERR=220 THEN PRINT "TOO LARGE"
IF ERL=35 THEN RESUME 30
RESUME 10
END
```

If you want to use an error message that regular BASIC already has, you can use ERROR *n*, where *n* is a defined code for an error message. For example, try different values for T in this program (such as T=8 or T=15).

```
REM ERROR2
T=9
ERROR T
END
```

EXP(n)

EXP(*n*) calculates the exponential function, which is the mathematical number *e* raised to the *n*th power, where *e* is approximately 2.718281828.

```
PRINT EXP(2)
EE=EXP(A)
```

FIELD

FIELD is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

FILES

FILES prints a directory of files on a disk without affecting your BASIC program. If you want to see a directory of a different disk, use FILES "DF0:". If you want to see a subdirectory of a different disk, use FILES "DF0:title".

```
FILES
FILES "DF0:"
FILES "DF0:BasicDemos"
```

FIX(*n*)

FIX(*n*) truncates the fractional part of a number and returns the whole number part without rounding. FIX(*n*) is similar to INT(*n*), but INT(*n*) takes the next lower number, or the number to the left on the number line, so for negative numbers the number is the next lower number. FIX(*n*) simply truncates. CINT(*n*) converts to an integer by rounding, and DEFINT also rounds.

```
PRINT FIX(2.5),FIX(3.22),FIX(4.75)
PRINT FIX(-2.5),FIX(-3.22),FIX(-4.75)
```

FN

See DEF.

FOR and NEXT

FOR and NEXT form statements that include a loop of instructions to be performed a given number of times. These are often called FOR-NEXT loops. Each FOR statement must have a corresponding NEXT statement, and each NEXT statement must have a preceding FOR statement. The form is

```
FOR c = a TO b [STEP s]
[loop statements]
NEXT [c]
```

where *c* is an index or counter and *a* is the first number (or numeric expression) assigned to the counter *c*. The computer executes the statements after the FOR statement down to the NEXT statement with c=a. At the NEXT statement, *c* is incremented by one, and control goes back to the statement just

after FOR. The loop is performed repeatedly until the limit *b* is exceeded.

If you wish to increment by a number other than one, use STEP. The increment may be positive or negative and may be a fraction. The NEXT statement may leave off the index. FOR-NEXT loops may be nested, but you do need to be careful that the FOR statements are matched properly with the NEXT statements. NEXT may combine the counters, such as NEXT *c,d.*

A FOR-NEXT loop with no statements between the FOR statement and the NEXT statement will simply be counting, which creates what seems like a pause in the program.

Several of the other sample programs in this chapter use FOR-NEXT loops of various forms. The following program illustrates several types of loops. The first loop uses T as a counter to print three blank lines. The FOR-NEXT loop with A as the counter uses a negative step size, so the numbers for the counter will be counting backward. The FOR-NEXT loop with B as a counter is nested in the A loop. This loop uses a step size of a fraction. Notice that your limit does not have to be the exact number for the counter; the loop stops when the limit is exceeded.

```
REM FOR
FOR T=1 TO 3
  PRINT
NEXT T
FOR A=6 TO 2 STEP -2
  FOR B=1 TO 3 STEP .4
    PRINT A;"*";B;"=";A*B
  NEXT B
NEXT A
END
```

FRE(*n*)

FRE(−1) returns the number of bytes of free memory available in the entire system. FRE(−2) returns the amount of stack space that has never been used. If *n* is not −1 or −2, the function returns the amount of free memory in BASIC's data segment. You may put either a constant or a variable in for the expression *n*. With the cursor in the Output window, you may type PRINT FRE(−1) to see how many bytes are left. You may want to check the size of a program by printing FRE(−1) before and after it is written or loaded.

```
PRINT FRE(X)
PRINT FRE(0)
```

GET
See PUT.

GET#
GET# is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

GOSUB and RETURN
GOSUB *l*, where *l* is a line number or a label, tells the computer to GO to a SUBroutine starting at line *l*, then RETURN when it finds the command RETURN. When a RETURN is encountered, the program returns to the statement directly following the GOSUB statement. GOSUB is used when a procedure is required several places in a program. You can have a GOSUB within another subroutine. You may enter a subroutine at different points. To go back to the main program, there must be a RETURN statement.

```
REM GOSUB
GOTO MAIN
110 PRINT
120 PRINT "PRESS THE SPACE BAR TO CONTINUE."
130 WHILE INKEY$<>" ":WEND
  CLS
  RETURN

MAIN:
PRINT "HAVE INSTRUCTIONS HERE"
GOSUB 110
PRINT "FIRST PROBLEM INTRODUCED"
GOSUB 120
PRINT "ANSWER PRESENTED"
GOSUB 110
END
```

GOTO
GOTO *l*, where *l* is a line number or label, transfers program execution to the specified line rather than going to the very next statement. You may go to a previous statement, a later statement, or the same statement.

```
REM GOTO
GOTO FIRST
120 PRINT "THIS IS SECOND"
GOTO LAST
FIRST:
```

```
  PRINT "THIS IS FIRST"
GOTO 120
LAST:
  PRINT "THIS IS LAST"
  PRINT
  PRINT "PRESS CTRL-C TO STOP"
190 GOTO 190
END
```

HEX$ (*n*)

HEX$(*n*) returns a string representing the hexadecimal (base 16) equivalent of the decimal (base 10) number or expression *n*. When you work with machine language programs, you may want to use hex numbers. The PATTERN command also uses hex numbers. Hex numbers are preceded by &H and consist of the numerals 0–9 and letters A–F, which represent the next numerals. Hex constants must be in the range of 0 through FFFF.

```
REM HEX$
BEGIN:
PRINT "ENTER A DECIMAL NUMBER"
INPUT "TO BE CONVERTED";D
PRINT
PRINT "THE EQUIVALENT HEXADECIMAL"
PRINT "VALUE IS ";HEX$(D)
PRINT:PRINT
GOTO BEGIN
END
```

See also OCT$ for an example program.

IF

IF starts a conditional branching command. There are several forms:

IF *test* THEN *line*
IF *test* THEN *action*
IF *test* THEN *line1* ELSE *line2*
IF *test* THEN *line1* ELSE *action2*
IF *test* THEN *action1* ELSE *line2*
IF *test* THEN *action1* ELSE *action2*
IF *test* THEN *action1* ELSE IF ...

where *test* is a relational or numeric expression; *line*, *line1*, and *line2* are line numbers; and *action*, *action1*, and *action2* are

valid commands. The actions may be a series of commands separated by colons. Instead of line numbers, you can use GOTO *line label.*

For the IF-THEN statements, IF the test expression is true, THEN the program branches to the line specified or performs the specified action. If the test expression is not true, the program goes immediately to the next statement following the IF statement (next line).

For the IF-THEN-ELSE statements, IF the test expression is true, THEN the program transfers to the *line1* specified or performs *action1.* IF the test expression is not true, the ELSE command transfers the program to *line2* or performs *action2.*

The test expression may contain arithmetic operators, relational operators, or logical operators. The logical operators are NOT, AND, OR, XOR, IMP, and EQV. Given tests A and B, which can be either true or false, NOT reverses the truth.

For example, NOT A where A is true would return false.

AND between two tests requires that both A and B be true to return a true.

OR says that if either A or B or both are true, it will return a true.

XOR (exclusive OR) says either A may be true or B may be true, but not both.

EQV (equivalence) returns a true if both A and B are true or if both A and B are false.

IMP is implication. A IMP B returns a true if both A and B are true, if A is false and B is true, or if both A and B are false (returns false if A is true and B is false).

```
10 REM IF
20 A=4:B=8
30 IF B=2*A THEN 60
40 PRINT "B<>2*A"
50 GOTO 70
60 PRINT "B=2*A"
70 IF B>A THEN PRINT "B>A" ELSE PRINT "B<=A"
80 PRINT "TRY AGAIN? (Y/N)"
90 A$=INKEY$
100 IF A$="N" OR A$="n" THEN END
110 IF A$<>"Y" AND A$<>"y" THEN 90
120 PRINT
130 INPUT "A = ";A
140 INPUT "B = ";B
150 GOTO 30
160 END
```

See also AND, EQV, IMP, NOT, OR, XOR.

IMP

IMP is a logical operator for implication; it is used to connect two or more relations and return a true or false value to be used in a decision such as an IF-THEN statement. A test operand is true if it is not equal to zero and false if it is equal to zero. Given two test conditions, A and B, the following is the result of IMP.

A	B	A IMP B
T	T	T
T	F	F
F	T	T
F	F	T

A sample statement is

```
IF X>Y IMP X>Z THEN GOTO XYSUB
```

See also IF.

INKEY$

INKEY$ detects whether a key has been pressed on the keyboard. This command is useful if you want your user to respond with just a one-key answer, such as yes or no (Y/N), a multiple-choice response, or a number or letter. The cursor must be in the Output window. The INKEY$ method has less chance of user error than INPUT because you can ignore unwanted keys.

The key pressed is not printed on the screen. INKEY$ does not wait for a key to be pressed. If no key is pressed when the INKEY$ is checked, then INKEY$ has the value of a null string.

```
REM INKEY$
BEGIN:
  PRINT
  PRINT "CHOOSE:"
  PRINT " 1 FIRST OPTION"
  PRINT " 2 SECOND OPTION"
  PRINT " 3 THIRD OPTION"
  PRINT " 4 END PROGRAM"
PICK:
  C$=INKEY$:IF C$=" " THEN PICK
  IF C$<"1" OR C$>"4" THEN PICK
  PRINT
```

```
  ON VAL(C$) GOTO ONE,TWO,THREE,FOUR
ONE:
  PRINT "** FIRST OPTION CHOSEN"
  GOTO BEGIN
TWO:
  PRINT "** SECOND OPTION CHOSEN"
  GOTO BEGIN
THREE:
  PRINT "** THIRD OPTION CHOSEN"
  GOTO BEGIN
FOUR:
  CLS
END
```

INPUT

INPUT allows the user to enter something as the program is being run. The computer will receive user input until the RETURN key is pressed. The input must not contain a comma, because a comma implies more than one value in the INPUT statement. The LINE INPUT statement does allow commas (see LINE INPUT). If the variable name for the input value is numeric and the user enters nonnumeric characters, there will be an error message and the computer will wait for more input.

INPUT A	Receives number
INPUT B$	Receives string
INPUT "PROMPT";N$	Prints prompt in quotation marks
INPUT "PROMPT",N	Suppresses question mark

```
REM INPUT
PRINT "WHAT IS YOUR NAME?"
INPUT N$
PRINT:PRINT "HELLO, ";N$
PRINT:INPUT "ENTER A NUMBER";N
PRINT "NUMBER TIMES 2 =";N*2
PRINT
END
```

INPUT#

INPUT# is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

INSTR

INSTR is a function used to locate a certain letter or string within another string. There are two forms:

INSTR(*s1,s2*) returns the position of the first occurrence of string *s2* in string *s1*. The value returned is a number that tells at which character position *s2* starts. If *s2* is not found in *s1*, the value returned is zero. INSTR(*n,s1,s2*) starts with the *n*th character to search for string *s2* in *s1*.

In the following sample program, D$ is a string listing the abbreviations for the months. You enter the name of a month, M$. M$ is changed to the first three letters you enter, then INSTR searches D$ for the first occurrence of the month name and returns the month number. If you enter only two letters, *MA*, the month could be MAR or MAY, but the first occurrence is MAR for the third month.

```
REM INSTR
D$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
BEGIN:
PRINT
INPUT "ENTER A MONTH ";M$
M$=LEFT$(M$,3)
M=(INSTR(D$,M$)-1)/3+1
IF M>=1 THEN MONUMBR
  PRINT "NOT IN LIST"
  GOTO BEGIN
MONUMBR:
  PRINT "MONTH NUMBER =";M
  GOTO BEGIN
END
```

INT(*n*)

INT(*n*) returns the integer value of the numeric expression *n*. An integer function considers the integer to be the greatest whole number less than the given value. If you think of a number line, the integer value is the first whole number to the left of the number *n*. If the number is positive, the value returned is the whole number with the decimal portion truncated. The number is not rounded. If the number is negative, the value returned is the whole number less than the given *n*.

```
PRINT INT(5.479),INT(5.688)
PRINT INT(0)
PRINT INT(-3.2),INT(-3.6)
```

See also CINT, CLNG, DEFINT, FIX.

KILL

KILL *filename* gets rid of or erases the specified file on the disk. "DF0:" specifies any disk in the disk drive 0.

```
KILL "program"
KILL "DF0:TEST"
```

LBOUND, UBOUND

LBOUND and UBOUND are described in a separate section entitled "Subprograms," following the dictionary of Amiga words.

LEFT$

LEFT$*(s,n)* is a string function that returns the left *n* characters of the string *s*, or the first *n* characters; *n* must be a number zero or greater. If the number *n* is greater than the length of the string *s*, only the string *s* is returned (no added blank spaces). The example

```
LEFT$("HI THERE,"2)
```

returns HI.

```
REM LEFT$
PRINT "ENTER A WORD"
INPUT W$
F$=LEFT$(W$,5)
PRINT "THE FIRST FIVE LETTERS ARE ";F$
END
```

See also the sample program for INSTR.

LEN*(s)*

LEN*(s)* is a function that returns the LENgth of a string *s*, or the number of characters in *s*. For example, the LENgth of a string "HELLO" is 5.

```
REM LEN
PRINT "ENTER A WORD OR PHRASE."
INPUT S$
PRINT "THE LENGTH OF YOUR INPUT IS"
PRINT LEN(S$);"CHARACTERS."
PRINT
END
```

LET

LET is used to assign values of expressions to variables in a program. The word *LET* is optional with this version of BASIC and can be omitted.

LET A=6 assigns the value of 6 to the variable name A. Another way to write the command is A=6.

```
LET B=54
LET X=X+Y
```

LIBRARY

LIBRARY is used to open a machine language subprogram from BASIC. See Chapter 5 for examples of how to use LIBRARY "graphics.library".

LINE

The LINE command is the basic drawing command to draw a line from one point to another, where the points are specified in coordinates. The coordinates are the number of pixels from the upper left corner of the screen. An *(x,y)* point of (10,20) would be a point 10 pixels to the right and 20 pixels down. The following example program shows several forms and options in the LINE command. The basic LINE command is to draw from the first point to the second:

```
LINE (x1,y1)-(x2,y2)
```

In the program below, lines 2 and 3 illustrate this basic form. Line 4 lists just the second point. The computer will start drawing from the second point in line 3 to the point on line 4.

Line 5 shows how you can specify a color after the points. The statement on line 6 illustrates the B option, which draws a box using the first point as the upper left corner of the box and the second point as the lower right corner of the box. The coordinates for any two opposite corners may be specified.

The BF option fills in the box with the specified opposite corners. The default value is a solid box. The fill may be changed with the PATTERN command (see also PATTERN). If you do not use a color number, you still need to use the right number of commas to indicate the B or BF option.

```
1 REM LINE
2 LINE (20,20)-(60,50)
3 LINE (30,60)-(70,85)
4 LINE -(80,70)
```

```
5 LINE (90,90)-(140,80),2
6 LINE (20,110)-(40,130),3,B
7 LINE (50,120)-(75,150),2,BF
8 LINE (90,115)-(110,140),,BF
9 END
```

With just the LINE graphics command, you can create beautiful designs. Draw the lines in certain patterns or in a certain sequence. The following program draws lines using three nested FOR-NEXT loops.

```
REM LINES
X1=320:Y1=0:X2=320:Y2=199
M=X1:N=X2
FOR J=1 TO 5
  FOR C=0 TO 3
    FOR D=1 TO 8
      LINE(X1,Y1)-(X2,Y2),C
      LINE(M,Y1)-(N,Y2),C
    X2=X2+5
    N=N-5
NEXT D,C,J
END
```

Now try the program when you add M=M+5 to the line with N=N−5. Try the program when you also add X1=X1−5 to the line with X2=X2+5. Try some graphics with the B and BF options as well.

LINE INPUT

LINE INPUT allows any character (except RETURN) that is on the keyboard to be input into a string. You may include a prompt in quotation marks like the regular INPUT statement.

On a regular INPUT statement asking for one item, if you enter a comma you will get the "Redo from start" message. LINE INPUT accepts the comma as part of the string. LINE INPUT allows only one string variable.

Leading blank spaces on INPUT are ignored, but with LINE INPUT they are part of the string. INPUT prints a question mark, but LINE INPUT does not.

```
REM LINE INPUT
INPUT W$
PRINT W$
LINE INPUT X$
PRINT X$
END
```

LIST

LIST lists or reprints your program in the List window. It is equivalent to the Show List option from the Windows menu. There are several forms of the command that are acceptable if you use line numbers or labels.

LIST lists the complete program. If you have a long program, the List window will show the section of the program where you were last working.

LIST *line* lists from line number or label. The line specified will appear at the top line of the List window.

```
LIST 100-200
LIST -200
LIST 200-
LIST BEGIN
```

LLIST

LLIST is just like LIST, but prints the listing on a printer.

LOAD

LOAD *filename* loads or reads into BASIC memory the specified file in quotation marks from disk into the computer. If you are using a different disk, you need the DF0: specification or the title of the disk. When you put programs in subdirectories or folders, you must enter the filename with the subdirectory or folder name (LOAD *subdirectory/filename*).

If you add ,R the program will load and run. Here are some examples of the LOAD command:

```
LOAD "STATES"
LOAD "DF0:TYPE1"
LOAD "REGENA:NOTES"
LOAD "BOOK:LOAN.BAS"
LOAD "DF0:BasicDemos/MUSIC",R
```

LOC

LOC is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

LOCATE

The LOCATE *r,c* command is used as an efficient way to print at a certain place on the screen or to relocate the cursor rather than printing blank lines and TABulating to a certain column.

The first parameter listed is the row number, *r*, and *c* is

the column number to start printing; (1,1) is in the upper left corner of the screen.

LOCATE 5,10:PRINT "MESSAGE"

would start printing MESSAGE in the fifth row from the top and in column 10.

LOF

LOF is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

LOG(*n*)

LOG(*n*) returns the natural logarithm of *n*, or log of *n* with the base *e*. Remember that the argument or expression *n* must be greater than zero or you will get an illegal function call error message. The logarithm and exponential functions are inverses:

X=LOG(EXP(X)) and X=EXP(LOG(X))

Example statements are

```
PRINT LOG(2)
X=LOG(Y-2)
IF LOG(N)<0 THEN 700
```

LPOS(*n*)

LPOS(*n*) returns the position (column number) of the printhead of the printer. The value of *n* does not matter.

```
IF LPOS(0)>60 THEN LPRINT CHR$(13)
```

LPRINT

LPRINT may be used just as the PRINT statement, but sends the printing to a printer. You may use TAB and SPC, constants, and variables. You may also use LPRINT USING with the same format specifications as PRINT USING.

```
LPRINT "HELLO"
LPRINT TAB(10);"NAME";SPC(20);"PHONE"
```

LSET

LSET is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

MENU, MENU ON, and MENU RESET

Push the right mouse button and move toward the top of the screen. The top highlighted line will change and menu titles

will appear. As you touch a title, a menu will drop down with several options. As you move the mouse downward, the subtitles will be highlighted. To make a selection, you release the right mouse button.

When you work with Workbench or with Amiga BASIC, these menus are set, but from BASIC you can put in your own menus so that as you are running your own program the user has menus for your program.

The form for creating a menu is

MENU ***menu-id,item-id,state [,title]***

The *menu-id* is the position of the menu selection on the menu bar and can be a number from 1 through 10. If you select 1, for example, your menu will replace the first (leftmost) menu. In the following sample program, we'll replace the second menu with our own custom menu.

The *item-id* is the relative position of the selected item in a menu and can be a value from 0 through 20. Item-id 0 refers to the entire menu. The other numbers are for the choices under the main topic.

The *state* is 0 to disable, 1 to activate, or 2 to activate and place a checkmark. The *title* is a string for the title of the item chosen. These titles will be what appears when the mouse moves to the menu bar.

MENU ON enables event trapping or use of the ON MENU GOSUB statement. You may choose to use the ON MENU GOSUB method or the method shown below.

There are two functions involved with MENU. MENU(0) is similar to INKEY$ and is reset to zero every time it executes. It returns a number which corresponds to the number of the last menu bar selection made or which main menu has been chosen.

MENU(1) returns a number of the last menu item chosen.

MENU RESET restores the original Amiga BASIC menu bar. In this case we are replacing the Edit menu, but when the program ends, MENU RESET puts back the Edit menu.

The following program illustrates how these MENU statements work. The second main menu will be replaced by our custom menu. It will have the main title of FACTORS with five subchoices. MENU ON activates the event trapping, so we can detect when the menu has been chosen. MENU(0) returns a number of the menu chosen. We will ignore the other menus and act only if our new menu is chosen, Menu 2.

MENU(1) returns which item was chosen in the list. When you choose to end the program, MENU RESET restores the original menu of Amiga BASIC.

```
REM MENUS
MENU 2,0,1,"FACTORS"
MENU 2,1,1,"All Factors"
MENU 2,2,1,"Prime Factors"
MENU 2,3,1,"Lowest CM"
MENU 2,4,1,"Greatest CF"
MENU 2,5,1,"End Program"
MENU ON

ACTIVE:
M=MENU(0)
IF M<>2 THEN GOTO ACTIVE
CHOICE=MENU(1)
ON CHOICE GOSUB AL,PRIME,LCM,GCF,E
GOTO ACTIVE

AL: PRINT "ALL FACTORS"
RETURN

PRIME: PRINT "PRIME FACTORS"
RETURN

LCM: PRINT "LOWEST COMMON MULTIPLE"
RETURN

GCF: PRINT "GREATEST COMMON FACTOR"
RETURN

E: PRINT "END PROGRAM"
MENU RESET
END
```

See also ON MENU.

MERGE

MERGE is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

MID$

MID$*(s,n,m)* is a string function that returns a segment of string *s* starting with character number *n* and *m* letters long. For example,

```
MID$("THIS IS AN EXAMPLE",6,5)
```

would return IS AN. Here are some other example statements:

```
A$=MID$(X$,J,3)
PRINT MID$(N$,2,6)
IF MID$(A$,X,Y)="CLUE" THEN GOSUB 500
```

MKI$, MLK$, MKS$, MKD$

These functions are described in a separate section entitled "File Processing," following the dictionary of Amiga words.

MOD

a MOD *b* returns the remainder when the numeric expression *a* is divided by *b*. For example, 11 MOD 4 returns the value 3. Eleven divided by 4 equals 2 with a remainder of 3 so the value returned is 3. Here are some example statements.

```
PRINT 11 MOD 4
IF X MOD Y <>0 THEN PRINT "NOT"
```

MOUSE

MOUSE statements are used when you are using the mouse arrow to receive input rather than the keyboard. An arrow appears on the screen. As you move the mouse, the arrow moves in that direction. There are several MOUSE commands and functions in Amiga BASIC that relate to the mouse and the position of the arrow. The MOUSE ON statement activates event trapping of the pressing of the left mouse button. Related to this command is the ON MOUSE–GOSUB statement which directs the program for events.

MOUSE OFF disables the ON MOUSE event trapping.

MOUSE STOP suspends mouse event trapping until the next MOUSE ON statement is executed. With MOUSE STOP, event trapping continues but the ON MOUSE–GOSUB statement is not executed. Event trapping does slow program execution; therefore, if the program no longer needs to read the button, MOUSE OFF is preferred to MOUSE STOP.

The MOUSE*(n)* functions are listed in your Amiga BASIC manual in detail. This function returns values that indicate whether the left mouse button was pressed and information about the position of the arrow. The function parameter *n* may be a number from 0 through 6.

MOUSE(0) returns the status of the left mouse button.

0 indicates that the button is not currently down and has not been pressed since the last MOUSE(0) function call.

1 indicates the button is not currently down but has been pressed once.

2 indicates the button has been pressed twice.

−1 indicates the button is being held down after clicking once—which usually means the mouse is moving.

−2 indicates the button is being held down after clicking twice.

MOUSE(1) is the x coordinate (horizontal) of the mouse pointer the last time the MOUSE(0) function was executed.

MOUSE(2) is the y coordinate.

MOUSE(3) returns the x coordinate the last time the left button was pressed before MOUSE(0) was called.

MOUSE(4) returns the y coordinate.

MOUSE(3) and MOUSE(4) are used together to locate the starting point of a mouse movement.

MOUSE(5) returns the x coordinate of the mouse pointer when MOUSE(0) was executed if the button was down when MOUSE(0) was called. If the button was up the last time MOUSE(0) was called, MOUSE(5) returns the coordinate of the pointer when the button was released.

MOUSE(6) returns the y coordinate. These two values are used to track the mouse as it is moved and to determine the coordinates where movement stops.

The following example programs illustrate the use of some of these MOUSE statements. The first short program simply checks the position of the mouse when the button is pressed. MOUSE(0) is used to check whether the button is pressed. The program stays at this line until the value of the function is not zero. When you press the left button, the value is no longer zero and the program continues. MOUSE(5) and MOUSE(6) are used to determine the ending x coordinate and y coordinate, and these coordinates are printed. You may move the mouse to various positions to see the coordinates when you press the left mouse button.

```
1 REM MOUSE1
2 WHILE MOUSE(0)=0:WEND
3 PRINT MOUSE(5);",";MOUSE(6)
4 GOTO 2
5 END
```

Following is a short program that shows how you can draw by pressing the left button and moving the mouse

around. Line 20 defines X and Y to be integers using DEFINT. Line 30 waits until the left mouse button is pressed. When the button is pressed, line 40 checks the current X coordinate with MOUSE(1) and the current Y coordinate with MOUSE(2) and returns the values X and Y. These coordinates are used in the PSET command in line 50 to turn on a point—place a white dot on the blue screen. Line 60 branches back to line 30 to keep checking the mouse button.

```
10 REM MOUSE2
20 DEFINT X,Y
30 WHILE MOUSE(0)=0:WEND
40 X=MOUSE(1):Y=MOUSE(2)
50 PSET (X,Y)
60 GOTO 30
70 END
```

See also the sample program in PUT for moving an object with the mouse.

NAME

NAME *filename1* AS *filename2* changes the name of a disk file from *filename1* to *filename2.*

```
NAME "TEST" AS "TRIAL"
NAME "DF0:WORK" AS "DF0:DRAWING"
```

NEW

NEW erases the BASIC program currently in memory and allows you to start or load a new program. There will be no more statements stored in the computer. All numeric variables return to zero. If you have been working on a program and made changes, when you type NEW, the computer will first ask if you want to save the current program. You may use the mouse to select yes or no before you start your next program.

NEXT

NEXT is the last statement in a FOR-NEXT loop. NEXT increments the loop counter or index. If the index is greater than the limit in the FOR statement, program control goes to the statement following NEXT; otherwise, the loop is performed again (control goes to the statement following FOR). The index variable name on the NEXT statement is optional unless loops are nested.

```
NEXT
NEXT C
NEXT C,D
```

The following example program illustrates several forms of the NEXT statement and some nested FOR-NEXT statements.

```
REM NEXT
FOR M=1 TO 5:PRINT M:NEXT M
PRINT
FOR J=1 TO 2000:NEXT
FOR M=1 TO 5
 FOR J=1 TO 3
  FOR K=10 TO 20 STEP 2
   PRINT M*J*K
   FOR L=1 TO 1500:NEXT
NEXT K,J,M
END
```

See also FOR, STEP.

NOT

NOT is one of the logical operators in Boolean algebra and is often used in IF-THEN conditional statements.

NOT 0 = −1
NOT −1 = 0
NOT *n* = −*n*−1 (for any number)

NOT reverses the state of a test in an IF-THEN statement.

```
REM NOT
A=1:PRINT A,NOT A
A=0:PRINT A,NOT A
A=-1:PRINT A,NOT A
A=-56:PRINT A,NOT A
A=34:PRINT A,NOT A
X=5:Y=6
IF NOT (X<Y AND X*Y=30) THEN PRINT "NOT"
IF NOT X>Y THEN PRINT "SECOND"
END
```

OBJECT

There are a number of OBJECT words which are described in detail in your Amiga BASIC manual. They each have to do with operations and functions of objects. In the statements, *id* refers to the object identification number and corresponds with the ID in an OBJECT.SHAPE statement.

OBJECT.SHAPE *id,d* defines the shape, colors, and location of an object with the given ID number. The definition *d* is a string which describes the size, shape, and color. The definition is defined by using the Object Editor utility program which is on the same disk as Amiga BASIC. In using the Object Editor you will save the shape on disk. When you use the shape in a program, you can define the object with

OPEN *"filename"* FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
CLOSE 1

OBJECT.X *id,n* and OBJECT.Y *id,n* position the object in the Output window with the specified pixel coordinates. The functions of OBJECT.X*(id)* and OBJECT.Y*(id)* return the current x and y coordinates of the upper left corner of the object.

OBJECT.VX *id,n* and OBJECT.VY *id,n* define the speed of the specified object in the x and y directions. The speed *n* is expressed in the number of pixels per second. The functions OBJECT.VX*(id)* and OBJECT.VY*(id)* return the speed in the x or y direction, also in pixels per second.

OBJECT.AX *id,n* and OBJECT.AY *id,n* define the acceleration of an object in the x and y directions. Here, *n* is the value of the acceleration rate in numbers of pixels per second per second. Acceleration is the change in velocity per time.

OBJECT.CLIP *(x1,y1)-(x2,y2)* defines a rectangle with opposite corners at coordinates *(x1,y1)* and *(x2,y2)*. Objects cannot be drawn outside this rectangle. The default value is the current Output window.

OBJECT CLOSE *[id][,id][...]* is used when you no longer need an object. It frees the memory used by that object. If you do not specify an *id* number, all objects in the current Output window are released.

OBJECT.HIT *id [,Me][,Hit]* determines collision objects for the specified object. If you do not specify *Me* and *Hit,* all objects collide with each other and the border. However, you can use this statement to allow objects not to record certain collisions. *Me* is a bit mask that identifies the object. Each object is assigned a different bit in *Me.* The border corresponds to bit 0. *Hit* is a bit mask that describes what other objects this object can collide with. Both masks are 16 bits. An object collides

with another object if the result of its *Hit* value ANDed with the other object's *Me* value is not zero. In the following example, object 1 can collide only with the border because its *Hit* value is 1. Object 2 can collide with objects 1 and 3 because its *Hit* value is the sum of the *Me* values of objects 1 and 3 (10=2+8). Object 3 can collide with the border and objects 1 and 2 (7=1+2+4).

OBJECT.HIT 1,2,1
OBJECT.HIT 2,4,10
OBJECT.HIT 3,8,7

OBJECT.ON *[id][,id][...]* makes the specified object visible. If an *id* is not specified, the current Output window will display all the objects. If the object has already been started with an OBJECT.START statement, it will begin to move again.

OBJECT.OFF *[id][,id][...]* makes the specified object invisible. The default if no *id* is listed is that all objects are invisible. If an object is started with OBJECT.START, this statement stops the object and prevents collisions.

OBJECT.PLANES *id [,plane-pick][,plane-on-off]* sets the blitter object's (bob) plane-pick and place-on-off masks, which can be integers from 0 through 255. The default values are established by the Object Editor program.

OBJECT.PRIORITY *id,n* sets priority of an object with relation to other objects. The priority number *n* is an integer. Higher priority numbered objects will be displayed in front of lower priority numbered objects.

OBJECT.SHAPE *id1,id2* is the format used to copy the shape of *id2* to *id1* creating a new object. This method is used in the sample program following.

OBJECT.START *[id][,id][...]* sets the object in motion. If *id* is not specified, all objects start moving.

OBJECT.STOP *[id][,id][...]* stops or freezes the motion of the object. If *id* is not specified, all objects stop.

To use these OBJECT statements, you need to just sit and experiment with them. Following is a sample program to get you started. First load the Object Editor utility program which is on the Amiga BASIC disk and run it. By the way, you may want to copy this program onto another disk that you use specifically for programming your own objects. When you run the program, you are able to design your own object. When you get the object the way you want, save it.

To try this sample program, use the Object Editor to design an object and save it as "DF0:BLOB". The DF0 indicates we are saving the object on a different disk from the original Amiga BASIC disk that we started the session with. Now exit the program and type in this program, or load it and run it. It uses the information you saved with the Object Editor to define the BLOB. You will need to use the disk on which you saved "BLOB".

```
REM OBJECT
OPEN "DF0:BLOB" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(L0F(1),1)
CLOSE 1

OBJECT.X 1,20
OBJECT.Y 1,50
SX=60:SY=50
OBJECT.VX 1,SX
OBJECT.VY 1,SY
OBJECT.ON

START: OBJECT.START

COLL: K=COLLISION(1)
IF K=0 THEN COLL
IF K=-1 OR K=-3 THEN SY=OBJECT.VY(1):OBJECT.VY
1,-SY:GOTO START
SX=OBJECT.VX(1)
OBJECT.VX 1,-SX
GOTO START
END
```

I used an OPEN statement to OPEN the file to get the information about the object you designed using the Object Editor program. The next line reads the information as a string with INPUT$(LOF(1),1). OBJECT.SHAPE 1 says to define shape number 1 with that previously saved file. CLOSE 1 closes this file that we will no longer need.

OBJECT.X and OBJECT.Y define where the object will start on the screen. SX and SY are the speed numbers which are used in OBJECT.VX and OBJECT.VY. OBJECT.ON makes our object visible, and OBJECT.START starts the object in motion. The routine at the beginning of COLL uses COLLISION(1) to see whether the object collided with the border. If K is zero, there is no collision and the object can keep moving. If K is −1 or −3, then the top or bottom border was hit

and the Y velocity needs to be changed; otherwise, the side borders were hit and the X velocity needs to be changed. GOTO START continues the program until you choose Stop from the Run menu or press CTRL-C. After you stop the program, you can get rid of the object by typing OBJECT.OFF.

OCT$ *(n)*

The OCT$*(n)* function returns the octal (base 8) value equivalent to the decimal number *n* (integer).

```
REM OCT$
INPUT "ENTER A DECIMAL NUMBER";N
PRINT
PRINT "THE OCTAL NUMBER IS ";OCT$(N)
PRINT "THE HEXADECIMAL NUMBER IS ";HEX$(N)
END
```

ON BREAK

Used to tell a BASIC program where to jump to when a break is encountered. You can break a program by selecting Stop from the Run menu, or by pressing either the Amiga and period keys, or the CTRL and C keys. ON BREAK requires a BREAK ON command.

```
REM ON BREAK
ON BREAK GOSUB STP
BREAK ON

WRTE:
PRINT "I'll KEEP PRINTING THIS UNTIL YOU"
PRINT "STOP ME"
PRINT
GOTO WRTE

STP:
BREAK OFF
FOR I=1 to 5
PRINT "NEXT TIME I'LL STOP FOR GOOD"
PRINT
NEXT
RETURN
END
```

See also BREAK.

ON COLLISION

See COLLISION.

ON ERROR

See ERL.

ON-GOSUB

ON *n* GOSUB line1,line2,line3,..., tells the computer to evaluate the numeric expression *n*, then branch to a subroutine starting with line1, line2, line3, and so on, depending on the value of *n*. The lines specified may be line numbers or line labels. If *n* is 1, the program will go to the subroutine starting at line1. If *n* is 2, the program will go to the subroutine starting at line2; if *n* is 3, GOSUB line3,.... Program control will execute the subroutine, then return to the line following the ON-GOSUB statement. See the MENU example program for an illustration of GOSUB using line labels. You may also use ON COLLISION GOSUB.

```
REM ON-GOSUB
BEGIN: PRINT
PRINT "CHOOSE:"
PRINT " 1 GAME ONE"
PRINT " 2 GAME TWO"
PRINT " 3 GAME THREE"
PRINT " 4 END PROGRAM"
PRINT

KEYPRESS:
C$=INKEY$:IF C$=" " THEN KEYPRESS
IF C$<"1" OR C$>"4" THEN 4
ON VAL(C$) GOSUB ONE, TWO, THREE, FOUR
GOTO BEGIN

ONE: PRINT "YOU CHOSE GAME ONE": RETURN
TWO: PRINT "YOU CHOSE GAME TWO": RETURN
THREE: PRINT "YOU CHOSE GAME THREE":RETURN
FOUR: PRINT "END PROGRAM":END
```

ON-GOTO

ON *n* GOTO line1,line2,line3,..., evaluates the numeric expression *n*, then branches according to the value of *n*. If *n* is 1 the program goes to line1; if *n* is 2 the program goes to line2; if *n* is 3 the program goes to line3,.... You may specify either line numbers or line labels or both.

```
REM ON-GOTO
BEGIN:
PRINT:PRINT "CHOOSE A NUMBER"
PRINT "1 2 3 4 5"
```

```
KEYPRESS:
A$=INKEY$:IF A$="" THEN KEYPRESS
IF A$<"1" OR A$>"5" THEN KEYPRESS
ON VAL(A$) GOTO ONE,TWO,THREE,FOUR,FIVE

ONE: PRINT "ONE"
TWO: PRINT "TWO"
THREE: PRINT "THREE"
FOUR: PRINT "FOUR"
GOTO BEGIN
FIVE: PRINT "END"
END
```

ON MENU

ON MENU jumps to a specific subroutine when MENU(0) is a nonzero value. MENU ON must have been executed for ON MENU to be active.

```
REM ON MENU
MENU 3,0,1,"For Fun"
MENU 3,1,1,"Quit"
ON MENU GOSUB QUIT
MENU ON

KEEPRINTING:
PRINT "Select Quit from the For Fun Menu to stop."
GOTO KEEPRINTING
QUIT:
MENU RESET
END
```

See also MENU.

ON MOUSE

See MOUSE.

ON TIMER

ON TIMER tells BASIC to jump to a subroutine every *n* seconds. For example, in the following program the subroutine PRINTIT will be executed every ten seconds. Note that TIMER ON must be executed to activate event trapping.

```
REM ON TIMER
ON TIMER(10) GOSUB PRINTIT
TIMER ON
W=0
WHILE W<200
PRINT W, "WE STOP AT 200"
```

```
PRINT "AND TAKE A BREAK EVERY TEN SECONDS"
W=W+1
WEND
STOP
PRINTIT:
PRINT:PRINT
PRINT "Another 10 seconds have gone by"
PRINT:PRINT
RETURN
END
```

OPEN

OPEN is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

OPTION BASE

OPTION BASE *n* sets the base for arrays; *n* may be either 0 or 1, and the default value (not specifying an OPTION BASE) is 0. This means that subscripts start at 0, for example, A(0), A(1), A(2),.... If you specify OPTION BASE 1, the subscripts start at 1 and memory is saved if you don't need the zero elements. The OPTION BASE statement needs to be executed before the DIMension statement that defines the arrays.

```
OPTION BASE 1
DIM A(16),B(16),A$(16,3),B$(16)
```

OR

Logical OR is used in IF-THEN statements to combine conditions. One condition OR the other OR both must be true for the THEN action to take place. More than one OR may be used.

OR differs from XOR in logical operators because XOR (exclusive OR) says one or the other condition, but not both, must be true for the THEN action to take place.

```
REM OR
1 PRINT
INPUT "ENTER A NUMBER ";X
INPUT "ENTER ANOTHER NUMBER ";Y
IF X<Y OR X<Y*2 THEN 6
PRINT "X>=Y OR X>=Y*2":GOTO 7
6 PRINT "X<Y OR X<Y*2"
7 PRINT
PRINT "TRY AGAIN? (Y/N)"
INPUT A$
IF A$="Y" OR A$="y" OR A$="YES" THEN 1
```

```
IF A$="N" OR A$="n" OR A$="NO" THEN 8
PRINT "SORRY, DON'T UNDERSTAND"
GOTO 7
8 END
```

PAINT

PAINT *(x,y),h[,b]* fills in an area with "paint," or a hue, *h*, starting from the point designated by coordinates *(x,y)* and going to the border color *b*.

The following short program illustrates the PAINT command. The WINDOW command defines a WINDOW with the type of 24 (so you can close it after you have run the program). The window's label is PAINT, and the size of the window is designated by coordinates. The next line draws a circle, then a line is drawn. Both of these are drawn with the color 3. The first PAINT command specifies coordinates in the top part of the circle and says to paint with color 3. The second PAINT command uses color 2, which is black, and paints to the orange border, color 3.

```
REM PAINT
WINDOW 2,"PAINT",(20,10)-(150,100),24
CIRCLE (65,55),40,3
LINE (20,10)-(150,100),3
PAINT (70,40),3
PAINT (40,50),2,3
END
```

PALETTE

PALETTE *n,f1,f2,f3* is used to define new colors. If you don't like the standard colors, you may change the colors using Preferences on the Workbench, or in BASIC you can use the PALETTE command. Think of the PALETTE command as using an artist's palette and mixing colors. For each of the four possible colors, or paint buckets, you can mix a combination of red, green, and blue. The first parameter, *n*, is a bucket number. The *f1*, *f2*, and *f3* numbers may be fractions from 0 through 1. Black is 0,0,0, or no colors, and white is 1,1,1, or a mixture of all colors. In order, the numbers are for the amount of red, green, and blue in your bucket.

The following program illustrates the PALETTE command by drawing four boxes of the four colors. PALETTE 0, or the background color, is changed to 0,0,0, which is black. PALETTE 1, which is the default drawing and printing color, is a

mixture of 0,0,1. PALETTE 2 is 0,1,0. PALETTE 3 is 1,0,0.

```
REM PALETTE
PALETTE 0,0,0,0
PALETTE 1,0,0,1
PALETTE 2,0,1,0
PALETTE 3,1,0,0
FOR C=0 TO 3
  LINE (C*20,80)-(C*20+20,120),C,BF
NEXT C
END
```

Now try some different fractional mixtures in the palettes.

```
PALETTE 0,.1,.8,.8
PALETTE 1,.3,.2,.4
PALETTE 2,.4,.1,.6
PALETTE 3,.8,.5,0
```

PATTERN

PATTERN is used to change the pattern of a line or an area that is filled with AREAFILL or the BF option in the LINE statement. The default patterns are solid. However, you may designate any pattern you like. You may use graph paper to draw out a pattern of filled-in squares and then convert each row to its hex equivalent. The pattern numbers start with &H, then continue with the hex pattern numbers.

The PATTERN command may specify a hex number for a line or an array for an area. The following short program illustrates how PATTERN can change from the solid pattern to a defined pattern. The array needs to be dimensioned before it is used. The PATTERN statement defines a line pattern, then the area array. Note that the line pattern also affects the cursor pattern. If you happen to use zeros in the hex pattern, the cursor may disappear. The LINE commands illustrate the pattern in a line and in a filled box. When you use NEW for another program, these patterns are no longer in effect.

```
REM PATTERN
DIM P%(3)
P%(0)=&HF0F0
P%(1)=&HA6A6
P%(2)=&H5555
P%(3)=&H3333
PATTERN &H5533,P%
LINE (0,90)-(100,90)
LINE (10,100)-(100,150),,BF
END
```

PEEK, PEEKL, PEEKW

Each of these functions returns a value store at the indicated memory location in the range 0–16777215.

PEEK returns an integer from 0 through 255.
PEEKL returns a 32-bit value (long-integer word).
PEEKW returns a 16-bit value (short-integer word).

```
REM PEEK
A$="COMPUTE!"
FOR I= 0 TO LEN(A$)-1
A=PEEK(SADD(A$)+I)
PRINT A,CHR$(A)
NEXT
END
```

```
REM ADDRESS OF EXEC LIBRARY
PEEKL(4)
END
```

```
REM PEEKW
A%=25
PRINT PEEKW(VARPTR(A%))
END
```

POINT

POINT *(x,y)* is a function that returns the color number of a point designated by the coordinates *(x,y)* in the current Output window. A value of −1 is returned if the point is not in the window. In the following example program, a rectangle of orange is drawn. The first point in question is within the rectangle and so returns a color number 3. The second point is not in the rectangle and returns the background color 0. This function may be used in calculations or in IF-THEN conditions.

```
REM POINT
LINE (20,20)-(10,150),3,BF
PRINT POINT(30,50)
PRINT POINT (10,180)
END
```

POKE, POKEL, POKEW

Each of these commands writes a value at the indicated memory location. The format is POKE *a,n* (POKEL *a,n;* POKEW *a,n*) where *a* is a memory address in the range of 0 through 16777215, and *n* is a value. *These commands should be used with care; changing memory can cause your system to crash.*

POKE writes an integer from 0 through 255.
POKEL writes a 32-bit value (long-integer word).
POKEW writes a 16-bit value (short-integer word).

POS(*n*)

POS(*n*) is a function that returns the current cursor column position; *n* is a dummy variable and may be anything. The first column is 1. The range of the returned numbers depends on the WIDTH command for the columns available. CSRLIN returns the row position. You may want to check the POS(*n*) and use the information in IF-THEN statements or later PRINT statements. In this sample program, the computer does some printing, then checks the position of the cursor.

```
REM POS
FOR J=1 TO 20
PRINT TAB(J);STRING$(J,42);
NEXT J
P=POS(N):C=CSRLIN
PRINT:PRINT "CURSOR ENDED AT";P;C
END
```

PRESET

PRESET should be thought of as Point-RESET. PRESET *(x,y)* resets the point designated by the x and y coordinates to the background color or erases a point that was previously set or drawn. Sometimes graphics can be drawn more quickly by filling in a solid area or drawing lines and circles, then resetting certain points—rather than setting many points.

The following program fills in a rectangle with the BF option of LINE and then resets several points to the background color. Only a few points are reset, so you will have to look carefully to see them.

```
REM PRESET
LINE (10,10)-(100,100),,BF
PRESET (30,40)
PRESET (80,40)
PRESET (55,55)
PRESET (54,65):PRESET (55,65)
PRESET (56,65):PRESET (55,66)
END
```

PRINT and PRINT USING

PRINT is the command to display something on the screen in text. The computer can do many operations, but you can't actually see them until something is output to the screen with a command such as PRINT. There are many forms that may be used in printing, and constants or variables may be printed.

LPRINT will print to the printer, and PRINT USING formats printing.

PRINT separators are the comma and the semicolon. The sample program illustrates the use of both of these.

To print a string, enclose it in double quotation marks. The sample program illustrates printing of variables with separators, plus the TAB and SPC functions. PRINT statements are used in many of the other sample programs.

```
REM PRINT
A=3:B=-5
A$="GRANT":B$="CHRISTINE"
PRINT A
PRINT A$;" ";B$
PRINT A$,B$
PRINT "123456789012345678 90"
PRINT TAB(5);"FIVE";TAB(12);B
PRINT TAB(2)A$SPC(6)B$
END
```

See also SPC and TAB.

PRINT USING *"f";list* prints an item or list according to a specified format *"f."* Different formats are available for numbers and strings. In numeric output, the # sign indicates placement of a digit. Numbers are right-justified and rounded rather than truncated. Combinations of these formats may be used, and you may add other characters to be printed within the quotes.

####.#	Prints numbers rounding to one decimal place.
$$##.##	Prints a dollar sign; two decimal places.
####	Prints an integer.
**###	Prints leading asterisks to fill the field.
###-	Prints trailing minus sign on negative numbers.
+###	Prints leading plus or minus sign.
#####,.#	Inserts commas every third place.

```
REM PRINT USING
A=123.432:B=-3.5:C=2:D=25.78
PRINT USING "###";A
```

```
PRINT USING "###";B
PRINT USING "###";C
PRINT USING "###";D
PRINT USING " $$###.##";A,B,C
PRINT USING "###-";A,B,C,D
PRINT USING "**####.#";A
PRINT USING "**####.#";B
PRINT USING "**####.#";C
PRINT USING "**####.#";D
END
```

PRINT USING can help make formatting printed strings much easier. Strings are left-justified. An exclamation point (!) indicates to use the first character of the string. The ampersand (&) indicates to print the complete string (no matter what length). To indicate characters for other lengths, use two back slashes (\ \ ). For 2 characters, use two back slashes, but for more characters, use spaces between the back slashes. The total length is two for the back slashes plus the number of spaces between. The sample program first uses 3 characters with " \ \", then 11 characters in the format.

```
REM PRINT3
N$(1)="KELLY":N$(2)="JENNIE"
N$(3)="ANGIE":N$(4)="BRIAN"
N$(5)="LAURIE"
FOR N=1 TO 5
 PRINT USING "!";N$(N)
NEXT N
FOR N=1 TO 5
 PRINT USING " \ \";N$(N)
NEXT N
FOR N=1 TO 5
 PRINT USING " \           \";N$(N)
NEXT N
PRINT USING "THE INITIAL IS !";N$(2)
END
```

PRINT#

PRINT# is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

PSET

PSET *(x,y),c* sets (turns on) a point with coordinates *(x,y)* and color *c*. PSET can be used for drawing detailed graphics using specified points. If no color is specified, the default is color 1,

which is white. Keep in mind that LINE and CIRCLE are a quicker way to set points in a certain pattern.

```
REM PSET
FOR J=10 TO 100 STEP 10
 FOR K=10 TO 30 STEP 10
  PSET (J,K)
 NEXT K
NEXT J
PSET (50,40),2
PSET (50,50),3
END
```

PTAB

PTAB*(n)* moves the print cursor to pixel *n*. Similar to TAB except that PTAB uses pixel positions, *n* can be any number from 0 through 32767.

PUT

PUT is used with GET to PUT a rectangle of graphics in a different place on the screen. This is handy to move an object. GET gets a rectangle of information or a picture from a specified area, then PUT places that rectangle back on the screen in a different place (the original stays there unless erased).

You need to use a DIMension statement to reserve an array large enough to keep track of the information in the rectangle you will be moving. GET is of the form

GET (x1,y1)-(x2,y2),A

where *(x1,y1)* are the coordinates of the upper left corner of the desired rectangle and *(x2,y2)* are the coordinates of the lower right corner. *A* is the array name given to this rectangle. The rectangle needs to be large enough to enclose the drawing you want to move.

PUT specifies the coordinates of the upper left corner where you want to put the array A. The form is

PUT (x,y),A

The following program illustrates the use of GET and PUT. The second line DIMensions the array D which will be used to contain the rectangle of information. The next line draws an ellipse, and the LINE command draws a filled box. Of course, you can use a more complex drawing. GET gets a

rectangle array D, then PUT copies that rectangle to an area with the upper left corner at (120,80).

```
REM PUT
DIM D(100)
CIRCLE (20,20),12,3,,,1.2
LINE (15,15)-(25,25),,BF
GET (10,5)-(30,35),D
PUT (120,80),D
END
```

To move an object on the screen (such as an icon in your own program), you can use a combination of the MOUSE functions and the GET and PUT commands. The following short program illustrates one way this can be done. A simple picture is drawn on the screen in the upper left corner. GET stores the information in the array D. The line labeled M: checks to see whether the left mouse button has been pressed. If not, the program stays at that line. When the button is pressed, the current x and y position is checked with MOUSE(1) and MOUSE(2). If it is different from the previous position by two pixels, PUT redraws the picture with PUT, and X and Y are reinitialized.

```
REM PUT2
DEFINT D,X,Y
DIM D(1000)
LINE (0,0)-(50,50),,BF
CIRCLE (25,25),20,3,,,1.5
CIRCLE (25,25),20,2,,,.3
GET (0,0)-(50,50),D
REM CHECK MOUSE
M:
 IF MOUSE(0)=0 THEN M
 IF ABS(X-MOUSE(1))>2 THEN P
 IF ABS(Y-MOUSE(2))<3 THEN M
P:
 PUT(X,Y),D
 X=MOUSE(1):Y=MOUSE(2)
 PUT(X,Y),D
 GOTO M
END
```

RANDOMIZE

RANDOMIZE will vary the "seed" used to choose random numbers. RND is the function used for random numbers, but

if you use RND in a program and repeatedly run it, you will notice that the numbers are the same each time. To get a true mix of random numbers, use RANDOMIZE. The RANDOMIZE command needs to come before the statement using RND.

If you use just the command RANDOMIZE by itself, the computer will stop and ask the user to input a number. If the number input is the same each time, the sequence will be the same; if the number input is different, the sequence is different.

RANDOMIZE *n* with a number specified uses the number *n* as a seed. Again, if *n* changes, the random numbers will change. To get random numbers each time the program is run without having to input a number or to change the *n* in the RANDOMIZE *n* statement, you can use RANDOMIZE TIMER because TIMER is a different number each time (depending on the time elapsed on the computer).

```
REM RANDOMIZE
RANDOMIZE TIMER
FOR D=1 TO 5
 PRINT INT(6*RND)+1
NEXT D
END
```

See also RND.

READ

READ is used in combination with DATA statements to assign constants to variables, either string or numeric. READ *v1* will read the first available data item and assign that value to the variable *v1*. The assignments start with the first item in the first DATA statement. The computer keeps track of a pointer that indicates which data item will be the next one when another READ is encountered. You may specify any number of variables in the READ statement. The variables in one statement may include strings, numbers, or a combination, but the data items must match.

READ will read through the data list item by item in the order they appear in the program unless a RESTORE statement is encountered, in which case data is restored and can be used over again, or data may start with a specified line.

For every READ item there must be a corresponding DATA item. Extra data items will be ignored. Although data items have been read and assigned, you will not notice any-

thing unless something is output to the screen using those variables. Here are some example statements:

```
READ X,Y,Z
READ T
READ N(J)
READ A$,A,B$
```

Sample program:

```
REM READ
FOR J=1 TO 8
 READ N(J)
 PRINT N(J)*2
NEXT J
DATA 5,8,-2,0,1,3,25,13
END
```

See also DATA and RESTORE.

REM

A REMark statement is a comment that is ignored by the computer. REM statements may be used to document your program or to provide explanations to someone who might look at your listing. While a program is running, you will not notice the execution of a REM statement. The first lines of all the sample programs in this chapter are REM statements indicating the title or the BASIC word illustrated. An apostrophe (') can also be used in place of REM.

```
REM TITLE
'THIS WILL BE IGNORED
REM Other characters are allowed &*^%
```

RESTORE

RESTORE is used in conjunction with READ and DATA statements, which assign constants to variables. Ordinarily, the READ statements read the data items in exact order, starting from the beginning of the program. The computer keeps track of which data item has been read and sets a pointer. The next READ statement uses the next data item, no matter where the DATA statements are placed in the program. RESTORE restores the pointer to the first data item in the first DATA statement for the very next READ statement, even if the data items had not been read. RESTORE *line* restores the data starting with the specified line (use either a line number or a line label). RESTORE or RESTORE *line* always restores for the very

next READ statement even though there may be in-between statements. Valid statements are

```
RESTORE
RESTORE 450
RESTORE IDAHO
```

In this sample program RESTORE 4 starts the data over with line 4 for the next READ statement. RESTORE starts the data over from the beginning for the next READ statement.

```
REM RESTORE
FOR J=1 TO 5
 READ N(J):PRINT N(J)
NEXT J
READ A,B,C:PRINT A,B,C
RESTORE 4
PRINT
FOR J=1 TO 6
 READ X(J):PRINT X(J)
NEXT J
RESTORE:PRINT
FOR J=1 TO 8
 READ X(J):PRINT X(J)
NEXT J
DATA 3,8,7,9,0,6
4 DATA 7,8,2,9,1
DATA 5,1,2,4,7,9,7
END
```

RESUME

RESUME *line* is used to return to a specified line after an ON ERROR GOSUB or ON ERROR GOTO statement has sent control to the error-handling routine. It indicates to resume error checking and normal program execution. You need to make sure the computer does not go to the RESUME statement before first registering an error.

See ERROR for a sample program.

RETURN

RETURN is used to transfer program control back to the main program flow from a subroutine. RETURN leaves the subroutine and returns to the program to the statement just after the related GOSUB or ON-GOSUB statement. You need to be careful that the computer does not encounter a RETURN statement without first executing a GOSUB statement or you will

get an error message. A subroutine may contain more than one RETURN statement if there are different exit points. A RETURN statement may end subroutines entered at different lines.

See also GOSUB and ON-GOSUB for sample programs.

```
REM RETURN
GOSUB SAMPLE
PRINT "DONE"
GOTO 9
SAMPLE:
 PRINT "SUBROUTINE"
 RETURN
9 END
```

RIGHT$

RIGHT$*(s,n)* is a string function that returns the last (or right) *n* characters in string *s; n* must be a number zero or greater. If the number *n* is greater than the length of the string *s*, only the original string is returned (no added spaces).

```
REM RIGHT$
A$="SAMPLE OF RIGHT$"
PRINT A$:PRINT
PRINT "RIGHT$(A$, )","STRING"
PRINT
PRINT TAB(9);4,RIGHT$(A$,4)
PRINT TAB(9);12,RIGHT$(A$,12)
PRINT TAB(9);0,RIGHT$(A$,0)
PRINT TAB(9);20,RIGHT$(A$,20)
X=8
PRINT TAB(10)"X",RIGHT$(A$,X)
END
```

RND

RND returns a random number which is a decimal fraction from 0 through 1. RND*(n)*, where *n* is any positive number, will give a random sequence of different numbers.

RND*(n)* where *n* is zero or negative will return the same number as the last random number chosen.

RND used without a number is the same as using a positive number.

Since RND returns a fraction and most uses are for whole numbers, you can use INT to get a random integer. The fraction first needs to be multiplied by a factor to get a number

greater than 1. For example, for random numbers less than 10, use INT(10*RND). Notice that this will yield numbers from 0 through 9. If you prefer numbers from 1 through 10, use INT(10*RND)+1. In general, for a range of random numbers between A and B, use INT((B−A+1)*RND)+A.

In the following sample program, the first eight numbers are the same each time the program is run. RND is used alone to show the fractions chosen. RND(0) repeats the last chosen random number. RANDOMIZE TIMER will change the seed each time, so a different sequence of random numbers is used each time the program is run. The next loop of random numbers chosen are integers from 0 through 9. The last loop chooses random numbers from 1 through 6.

```
REM RND
FOR J=1 TO 8
 PRINT RND
NEXT J
PRINT RND(0)
RANDOMIZE TIMER
PRINT
FOR J=1 TO 8
 PRINT INT(10*RND)
NEXT J
PRINT
FOR J=1 TO 5
 PRINT INT(6*RND)+1;
NEXT J
END
```

RSET

RSET is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

RUN

RUN is the command to start running a program or executing a program from the beginning. RUN *line* will start execution at the specified line number. A line label is not allowed in the RUN statement. RUN *filename* will load a program from the disk and start running it.

```
RUN
RUN 450
RUN "DF0:ALGEBRA"
```

SADD

SADD*(s)*, where *s* is a string, will return the address of the first byte of that string.

```
REM SADD
A$="This is a string"
B=SADD(A$)
PRINT B
PRINT CHR$(PEEK(B))
END
```

See PEEK for another sample program.

SAVE

SAVE is the command to save a program currently in memory onto a disk. If you type SAVE without a program name, the computer will show a dialog box and ask you to name the program. That name then appears on the window and later SAVE commands will not need the title. To save to a different disk, you will need the "DF0:" label.

```
SAVE "DICE"
SAVE
SAVE "DF0:COUNTING"
```

SAY

SAY *s[,a]* is used to say a string. You may specify an array *a* that sets the attributes of the speech. SAY may also be used with the TRANSLATE$ option. The Workbench disk and a lot of memory are required to use speech. Once the conditions of speech are defined, you can type something for the computer to say. With SAY TRANSLATE$ you may use a string in quotation marks within parentheses, or you may use a string variable. Another way to get the computer to speak is to use the SAY command using phonemes. A description of phonemes is given in the Amiga BASIC manual. Several of the vowel sounds use different spellings, so you will need to keep the chart handy. The phonemes must be available as shown on the chart or there will be an error message.

To set up the speech conditions array, you can read in parameters from a DATA statement or define each element. The parameters in order are as follows:

pitch—expressed in hertz, is a number between 65 and 320. The default is 110, which is a male speaking voice.

inflection—0 for using inflections and emphasizing syllables, or 1 for a robotlike monotone. The default is 0.

rate—a number between 40 and 400 words per minute. The default is 150.

voice—0 for male; 1 for female. Default is 0.

tuning number—the sampling frequency in hertz, which may range from a low of 5000 to a high of 28000. The default is 22200.

volume—a number from 0 for no sound through 64 as the loudest.

channel—combination of left and right channels and is a value from 0 through 11.

mode—0 for synchronous speech output or 1 for asynchronous.

control—used when the mode is 1; it may be 0 for saying one statement, then the next, 1 for canceling the previous statement, and 2 for immediately interrupting the first statement and executing the second one.

A sample program segment to define the speech array (which must be integer) is to use data:

```
FOR T=0 TO 8
 READ S%(T)
NEXT T
DATA 110,0,150,0,22200,64,1,0,0
```

Using the above lines to set up the array, add these lines for one method of speech:

```
SAY TRANSLATE$("HELLO"),S%
M$="THIS IS A TEST."
SAY TRANSLATE$(M$),S%
```

Or you might use the same definition lines above and use the following for the method using phonemes. You may recognize this speech as counting in French.

```
TEXT$="AHN DUH TWAA KAETR SEYNK
       SIYS SEHT WIYT NAHF DIYS"
SAY TEXT$,S%
END
```

SCREEN

SCREEN *id,w,h,d,m* defines a new screen with the specified *id* number. The width and height and depth are specified as the next three parameters, and *m* is the mode. The depth determines the number of colors it can hold, and the mode is the

screen resolution. SCREEN CLOSE *id* closes the specified screen.

The depth is a number from 1 through 5. Two to the "depth" power will be the number of colors available. A depth of 1 has two colors, 2 has 4 colors, 3 has 8 colors, 4 has 16 colors, and 5 has 32 colors. The screen width and height are expressed in pixels. The mode number determines low or high resolution, which is the width in pixels of 320 for low or 640 for high. It also determines whether the screen is interlaced or not, which is the number of horizontal lines appearing on the screen. The modes are

1 Low resolution, noninterlaced
2 High resolution, noninterlaced
3 Low resolution, interlaced
4 High resolution, interlaced

The screen *id* number is the same *id* number used as the last parameter in a WINDOW definition.

```
SCREEN 2,320,200,3,1
WINDOW 2,"Fun",(10,10)-(100,50),15,2
```

SCROLL

Scrolls a specified area of the Output window. SCROLL *(x,y)-(xx,yy),n1,n2*, where *(x,y)-(xx,yy)* define a rectangle, and *n1* indicates the number of pixels right, and *n2* the number of pixels down, to scroll the defined area.

```
REM SCROLL
CLS
PRINT "SCROLL THIS"
FOR DELAY=1 TO 700:NEXT
SCROLL(0,0)-(190,190),50,50
END
```

SGN(*n*)

SGN*(n)* is a function that returns the sign of the numeric expression *n*. The expression is first evaluated. If the number evaluated is positive, SGN*(n)* will be 1. If the number is negative, SGN*(n)* will be −1. If the number is zero, SGN*(n)* will be 0. This function is handy in determining relative positions, for example, in a game, or in determining direction from one position to another.

```
PRINT SGN(-3)
PRINT SGN(0)
PRINT SGN(55.67)
```

SHARED

SHARED is described in a separate section entitled "Subprograms," following the dictionary of Amiga words.

SIN(*n*)

SIN(*n*) is a function that returns the sine of the angle of *n* radians. The following program graphs the sine function.

```
REM SIN
LINE (0,100)-(640,100)
FOR X=0 TO 32 STEP .1
 Y=100-(40*SIN(X))
 LINE (X*20,Y)-(X*20,100)
NEXT X
END
```

SLEEP

The SLEEP command causes BASIC to wait for an event such as a keypress, button press, collision, or menu selection.

```
REM SLEEP
ON MOUSE GOSUB CHECKBUTTON
MOUSE ON
DONE=1
WHILE DONE=1
 PRINT "SLEEPING"
 SLEEP
WEND
PRINT "THANKS"
STOP
CHECKBUTTON:
 IF MOUSE(0)<>0 THEN DONE=0
 RETURN
END
```

SOUND

SOUND *f,d[,v][,c]* is the basic command to produce a musical tone; *f* is a frequency (pitch), *d* is duration, *v* is volume, and *c* is the audio channel from 0 through 3. The frequency is a number for the standard cycles per second (hertz) for a tone, such as 440 for A. The duration is a number for the length of time you want a tone to play. The volume may be a number from 0 through 255, where 255 is the loudest. The default value is 127.

I generally like to use a variable for the duration, then set

up notes in terms of that variable. For example, if T is a quarter note, T/2 is an eighth note and 2*T is a half note. Here is a sample program first playing notes without the volume parameter, then different volumes.

```
REM SOUND
T=10
SOUND 262,T
SOUND 330,T/2
SOUND 392,T/2
SOUND 523,2*T
SOUND 262,T,80
SOUND 330,T,240
SOUND 392,T,100
SOUND 330,T,180
SOUND 262,2*T,120
END
```

To hear more than one voice, you need to specify the channel number. Also, since it can take quite a bit of typing to list each SOUND command, try putting the frequencies in DATA statements. First, a duration is read that is a factor multiplied by the time variable. S is the channel number. Here's just the beginning of a tune.

```
REM SOUND2
T=4
AGAIN:
READ D:IF D=0 THEN GOTO FINISHED
FOR S=0 TO 3
 READ F
 SOUND F,D*T,150-S*20,S
NEXT S
GOTO AGAIN
DATA 2,466,392,311,156
DATA 2,622,392,311,233
DATA 2,622,466,392,196
DATA 2,784,466,392,196
DATA 0
FINISHED:END
```

SPACE$

SPACE$(*n*) is a string function that yields *n* number of spaces for use in combination with other strings. For example, SPACE$(3) is " ".

```
PRINT "HELLO"+SPACE$(5)+N$
```

SPC(n)

SPC(*n*) is used in a PRINT statement to print *n* number of spaces between items.

```
REM SPC
PRINT "123456789012345678 90"
PRINT "ONE";SPC(5);"TWO"
PRINT "THREE"SPC(8)"FOUR"
END
```

SQR(n)

SQR(*n*) is a numeric function which returns the square root of a numeric expression *n*, where *n* is zero or positive. The square root means that a number multiplied by itself will result in the number *n*.

```
PRINT SQR(144)
IF SQR(X)>Y THEN 20
PRINT SQR(X*X+Y*Y)
```

STEP

STEP is an optional word in FOR-NEXT loops. STEP *s* indicates the increment size for the loop index; *s* may be positive, negative, or a fraction. The default value for the step size is +1. In the statement

```
FOR C=L1 TO L2 STEP S
```

the index, or counter, C will start at L1 to perform the loop. When the word NEXT is executed, C is incremented by S (or decremented if S is negative). If the new C is greater than L2 (less than L2 if S is negative), program control goes to the statement immediately after NEXT; otherwise, the loop is performed again, starting at the statement after FOR (see also FOR). Several of the sample programs use FOR-NEXT loops with STEP specified. Valid statements are

```
FOR C=1 TO 10 STEP 3
FOR J=2 TO 12 STEP 2
FOR K=25 TO 21 STEP -1
FOR L=10 TO 20 STEP .5
FOR C=A TO B STEP S
```

STICK and STRIG

STICK(*n*) returns a value of 1 if the joystick is pushed down or to the right, −1 if pushed up or to the left, and 0 at all other times. The value of *n* indicates which joystick and direction:

STICK(0) Reports joystick A horizontal movement.
STICK(1) Reports joystick A vertical movement.
STICK(2) Reports joystick B horizontal movement.
STICK(3) Reports joystick B vertical movement.

To check the status of the button use STRIG*(n)*:

STRIG(0) Reads status of button on joystick A. Returns 1 if button has been pressed since last execution of STRIG(0); otherwise returns 0.
STRIG(1) Reads status of button on joystick A. Returns 1 if button is being pressed while STRIG(1) is being executed; otherwise returns 0.
STRIG(2) Reads status of button on joystick B. Returns 1 if button has been pressed since last execution of STRIG(2); otherwise returns 0.
STRIG(3) Reads status of button on joystick B. Returns 1 if button is being pressed while STRIG(3) is being executed; otherwise returns 0.

```
REM JOYSTICK
FOR I=0 to 10
 READ V$(I)
NEXT
WHILE STRIG(2)=0
A=STICK(2):B=STICK(3)
PRINT V$(A*2+B*3+5)
WEND
DATA NW,,N,W,NE,,SW,E,S,,SE
END
```

STOP

STOP is a command that will stop the program from executing any more statements. This command is equivalent to pressing CTRL-C or choosing Stop from the Run menu. A program that is STOPped can usually be CONTinued. A program stopped with END cannot be CONTinued.

STR$

STR$*(n)* is a string function that converts a numeric expression *n* from a number to a string for use in combinations with other strings or to use with other string functions.

```
PRINT STR$(23.5)
PRINT RIGHT$(STR$(N),2)
N$=STR$(N)
```

STRING$

STRING$(*n,c*) returns a string of *n* number of characters with the ASCII code *c* or specified as a single character in quotation marks. If you need to print a long string of characters, this method is easier to type.

```
REM STRING$
PRINT STRING$(25,"R")
PRINT STRING$(15,49)
FOR S=1 TO 10
 PRINT STRING$(S,42)
NEXT S
END
```

SUB

SUB is described in a separate section entitled "Subprograms," following the dictionary of Amiga words.

SWAP

SWAP *v1,v2* will exchange the values of two variables, *v1* and *v2*. Any type variable may be changed—integer, single-precision, double-precision, string—but both variables must be the same type. Example statements are

```
SWAP A$,B$
IF A(N)<A(N+1) THEN SWAP A(N),A(N+1)
```

SYSTEM

The command SYSTEM will cause the computer to exit BASIC and return to the Workbench screen with the window showing the contents of the disk you used to load BASIC. To get back into BASIC, you would have to load BASIC again.

TAB

TAB(*n*) is a function that is similar to the tabulator key on a typewriter. TAB(*n*) is used in a PRINT statement to move to a specified column *n* before printing begins. The columns are numbered starting with 1. Keep in mind that numbers allow one space for the sign, so positive numbers will actually be printed in the next column. The semicolon after the right parenthesis is optional. You may specify more than one TAB in a PRINT statement.

```
REM TAB
PRINT "123456789012345678901"
```

```
PRINT TAB(5);"FIVE"
N$="BRETT":X=7:Y=-4
PRINT TAB(15);X
PRINT TAB(15);Y
PRINT TAB(8)N$
END
```

TAN(*n*)

TAN(*n*) is a numeric function that returns the tangent of the angle *n*, where *n* is expressed in radians. Keep in mind that TAN can be undefined at certain points or can return very large or very small numbers.

```
PRINT TAN(.78)
X=TAN(THETA)
```

THEN

THEN is a word in the IF-THEN conditional branching statement. It can be followed by an action or a line number or line label. IF the test condition is true, the action following THEN will be executed. If a line number is listed, program control will go to that line. The IF statement can also contain the word ELSE after THEN. Several of the sample programs use IF-THEN conditional statements.

```
IF A THEN B=0
IF X*Y>100 THEN PRINT "LARGE"
IF X=Y THEN 30
IF SC=10 THEN GOTO WIN
```

See also IF.

TIME$

TIME$ returns the current time in a string in the form *hh:mm:ss*.

```
PRINT TIME$
```

TIMER

TIMER is a number representing time elapsed since the computer was turned on. The difference between two TIMER values can be used in a program to designate time. The time is in seconds. Following is a short program that illustrates the use of TIMER. The program will time how long it takes you to type in a message. Line 50 will BEEP to signal the start of the timing. Line 60 sets the variable T1 to TIMER. Line 70 is IN-

PUT to receive your typing. When you press RETURN, line 80 sets the variable T2 to the new value of TIMER. Line 100 prints the length of time, which is the difference between T2 and T1.

```
10 REM TIMER
20 PRINT "TYPE IN A MESSAGE THEN PRESS RETURN."
30 PRINT "START AT THE TONE."
40 FOR DELAY=1 TO 2000:NEXT DELAY
50 BEEP
60 T1=TIMER
70 INPUT MSG$
80 T2=TIMER
90 PRINT:PRINT
100 PRINT "THE TIME WAS";T2-T1;"SECONDS."
110 END
```

See ON TIMER.

TO

TO is a word used in the FOR statement. FOR c=a TO b, where *c* is a counter and *a* and *b* are limits, is the first statement of a FOR-NEXT loop.

See also FOR.

TRANSLATE$

See SAY.

TRON and TROFF

TRON will trace each statement of a program as it is executed. TROFF turns off the trace. This is an excellent debugging tool.

UBOUND

UBOUND is described in a separate section entitled "File Processing," following the dictionary of Amiga words.

UCASE$

UCASE$*(s)* is a string function used to convert a string to uppercase. It is handy in programs where you can convert input strings to uppercase for testing correct answers. The input string can be in either uppercase or lowercase and UCASE$ will convert it to all uppercase.

```
REM UCASE$
INPUT "ENTER A WORD ",W$
W$=UCASE$(W$)
```

```
PRINT W$
END
```

VAL

VAL return a value of a string. This is the inverse of STR$.

```
REM VAL
V1$="1"
V2$="2"
PRINT V1$,V2$
V1=VAL(V1$):V2=VAL(V2$)
PRINT V1,V2
PRINT V1+V2
END
```

See also STR$.

VARPTR

VARTR reports the address of the first byte of a variable.

```
REM VARPTR
A%=100
B=VARPTR(A%)
PRINT B,PEEKW(B)
END
```

See also PEEK and SADD.

WAVE

WAVE *c,a* is a command that defines a sound wave pattern for channel *c* with an array *a*. The default value is the sine wave, or WAVE 0,SIN, which indicates a pure tone. To make the tone sound different and perhaps get "noises" instead or sounds of different instruments, you can change the wave. The channel number *c* may be from 0 through 3. The array is an integer array of 256 numbers (elements 0–255). You will need a DIMension statement near the beginning of the program to reserve space. For example, if we call our array W, we can use

```
DEFINT W
DIM W(255)
```

Now you can put different numbers into the W array. For example,

```
FOR C=0 TO 255:W(C)=INT(80*RND):NEXT C
```

or you may make calculations (perhaps a trigonometric function) for each value of W(C), or you may read values from

DATA. Now use the WAVE command to set the waveform for a particular channel:

```
WAVE 0,W
WAVE 1,W
```

You may want to set each channel to a different array.

Next, use the SOUND command to hear how your wave values affected the sound.

```
SOUND 440,20,128,0
```

```
REM WAVE
DEFINT W
DIM W(255)
WAVE 0,SIN
SOUND 440,20,128,0
RANDOMIZE TIMER
FOR C=0 TO 255
  W(C)=INT(80*RND)
NEXT C
WAVE 0,W
SOUND 440,20,128,0
END
```

WHILE and WEND

WHILE and WEND form a loop that executes WHILE a certain condition is true. WEND is the last statement of the loop.

```
REM WHILE
X=0:Y=0:X2=0:Y2=199
WHILE Y<199
 LINE (X,Y)-(X2,Y2)
 Y=Y+3:X2=X2+10
WEND
X=0:Y=0:X2=600:Y2=0
WHILE Y2<199
 LINE (X,Y)-(X2,Y2)
 X=X+10:Y2=Y2+3
WEND
END
```

WIDTH

WIDTH *n* designates how many columns may be printed on the Output screen. You can use WIDTH to format printing or to keep printing within certain columns to remain visible. WIDTH does not change the size of the printing. You can add spaces to avoid splitting words inappropriately.

```
REM WIDTH
WIDTH 15
PRINT "TRY PRINTING THIS SENTENCE WITH 15
COLUMNS."
WIDTH 25
PRINT "NOW TRY THIS SENTENCE TO SEE HOW IT DOES."
END
```

WINDOW

WINDOW is the command to allow you to define your own windows. The basic command is WINDOW *id*, where *id* is the identification number. The Output window that appears while you are in BASIC is window 1, so for your own windows you should specify a number greater than 1.

You may add more information about the window. In order, the window may have a title expressed in quotation marks, size with (x1,y1) and (x2,y2) coordinates, a type which sets up how much the user can do with the window, and a screen ID which can be a value from 1 through 4. When you use the WINDOW statement, a new Output window is created and displayed and brought to the front of the screen. NEW gets rid of the windows you created.

The title will appear in the top bar of the window and is a string expression. The type is a number from 0 through 31:

1 Sizing gadget appears in the lower right side of the window to permit changing the window size.
2 Title bar can be used to move the window.
4 The window can be moved in relation to other windows, front and back, and that gadget appears in the upper right corner.
8 Close gadget allows the closing of the window.
16 Allows you to cover the window with another window without losing the contents.

To specify the type, add together two or more of these values.

WINDOW CLOSE *id* is the command to make a window invisible.

WINDOW OUTPUT *id* names the window for current output without moving the window to the front—direct output can go to a window that is behind another.

The following program illustrates some of the options available in the WINDOW command. It sets up three windows, then puts something in each window. Notice that the types are different, so different gadgets are in the corners.

```
REM WINDOWS
WINDOW 2,"Printing",(10,10)-(250,50),14
WINDOW 3,"Lines",(265,15)-(500,65),7
WINDOW 4,"Circles",(15,65)-(300,180),6
WINDOW 2
COLOR 3,2:LOCATE 3,5
PRINT "This is Window 2"
WINDOW 3
X2=10
AGAIN:
LINE(0,0)-(X2,100)
X2=X2+10
IF X2<400 THEN GOTO AGAIN
WINDOW 4
X=15:Y=10
FOR I=1 TO 9
 CIRCLE (X,Y),20
 X=X+20:Y=Y+10
NEXT I
END
```

WRITE

WRITE outputs data to the screen. WRITE alone leaves a blank line. WRITE is similar to PRINT (see PRINT), but WRITE separates items in a list by a comma. Also, WRITE puts quotation marks around strings. A positive number is not printed with a space before it.

```
REM WRITE
WRITE "HELLO"
X=5:Y=-7
WRITE X,Y
N$="REGENA"
WRITE N$
END
```

XOR

XOR is a logical operator name for exclusive OR. If there are two conditions A and B, A XOR B will return a true if only one of the conditions is true. OR returns a true if one condition is true or the other condition is true or both conditions are true.

See also OR and IF.

```
IF A=B XOR B=C THEN PRINT "OK"
IF N$="BOB" XOR SCORE=10 THEN GOTO WIN
```

Subprograms

Amiga BASIC has the capability of using subprograms. Subprograms are similar to subroutines contained within programs, but use variables that are separate (local) from those in the main program. Subprograms may be appended to a regular program and often may be routines that you want to add or merge with others without rewriting for compatibility each time.

To use a subprogram, you CALL it from the main program and pass variables in the *argument.* Here is a very simple example of a subprogram to illustrate the commands involved. The subprogram starts with the word SUB and has a title with the passed variables listed in parentheses. It ends with END SUB.

```
REM SUB
A=5:B=6
CALL MULTIPLY (A,B)
PRINT "BACK TO MAIN"
END

SUB MULTIPLY (X,Y) STATIC
Z=X*Y
PRINT Z
END SUB
```

The subprogram is called MULTIPLY and uses the variables X and Y. It prints the product, then has END SUB to return it to the main program. STATIC indicates that values of the variables will remain the same—they cannot be changed by actions outside the subroutine.

The main program assigns values to variables called A and B; it then calls the subroutine using the names for the variables in order A and B which will correspond to X and Y. When the subprogram has been executed and returns to the main program, the message BACK TO MAIN is printed and the program ends.

In a statement in the subprogram you may also specify SHARED with a list of variables specifically declared to be SHARED—altered by parts of the program outside the subprogram. If you use a SHARED statement, the variables do not need to be listed in the subprogram name:

```
SUB MULTIPLY STATIC
SHARED A,B
```

Arrays may be used in the arguments for subprograms. They are indicated with parentheses, such as R( ). You may include a number within the parentheses which would indicate the number of dimensions in the array, such as D(3) for a three-dimensional array. It does not indicate the number of elements in the array.

You may need the EXIT SUB command to leave the subprogram, but not at the last statement of the subprogram which is END SUB. For example, an IF-THEN condition could exit the subprogram with EXIT SUB.

LBOUND and UBOUND are functions that are especially useful in subprograms. They return the lower bound (0 or 1) and upper bound of a specified dimension of an array. For a one-dimensional array, you may specify the array name. For an array with more dimensions, you also need to specify the dimension you need. A general subprogram can be written, then LBOUND and UBOUND used as the specific limits for a particular program. For example, suppose you have arrays A and B, where A has one dimension and B has three, and you need to call a subprogram.

```
CALL SPECIAL (A( ),B( ))
...
SUB SPECIAL (X( ),Y( )) STATIC
 FOR J=LBOUND(X) TO UBOUND(X)
  FOR K=LBOUND(Y,2) TO UBOUND(Y,2)
....
```

The last loop varies K from the lower bound of the second dimension of the array Y to the upper bound.

File Processing

File processing concerns using other devices within the program. Often such input and output is abbreviated I/O. For examples in this chapter, we'll use the disk drive for input and output. Other devices, however, are available for other purposes—SCRN: is the screen (current Output window) for output, KYBD: is the keyboard for input, LPT1: is the line printer for output, and COM1: is the communications (serial) port for input or output.

In the syntax for the following commands, *f* is a file number, which may be from 1 through 255; *fname* is a filename or specification which may be up to 255 alphanumeric characters,

but not a BASIC reserved word. The filename may be a variable string. The filename may include the path or the volume name, such as "DF0:BasicDemos/music" or "BOOK:NOTES". *vlist* is the list of expressions or variables that will be input or output.

In working with program files (or simply programs), the most common commands are SAVE and LOAD. The general format for SAVE is

SAVE *fname*

but you may also include **,A** for saving in ASCII format and/or **,P** for saving in protected format (you won't be able to list or copy the program). For example,

```
SAVE "DF0:ALGEBRA",P
SAVE "UTILITIES:SORT1",A
```

If you're working on a program and save every so often, you may use just the command SAVE. The dialog box will ask if you want to save the program with the same name, and you may choose to do so (or rename the program as something else).

To bring a new program into memory, use the command LOAD with a filename in quotation marks. You may also keep the current program and merge it with a previously saved program with the MERGE *fname* command. A program loaded in with MERGE must have previously been saved with the ASCII format.

Within a program you can chain programs with the commands

CHAIN *fname*
CHAIN MERGE *fname*
CHAIN MERGE *fname,line,***ALL**

ALL indicates that all variables will retain their values from one program to the next. If you do not use ALL, the first program may use COMMON *vlist* to indicate which variables may be passed to the next program. The CHAIN MERGE command may also include DELETE *line-line* to delete lines after the programs are merged.

FILES is used to see the names of the files you have on your disk. You may also use FILES *pathname,* where *pathname* indicates the path or volume name or subcategory name.

```
FILES
FILES "DF0:"
```

You may define a pathname or change it with CHDIR (change directory) so that you can use a different disk. For example, to change to the current disk in the disk drive and not have to type "DF0:" each time, use

```
CHDIR "DF0:"
```

Sequential Files

There are two types of data files that can be used with the Amiga—sequential and random access. The sequential files are easier to understand and to work with, but the random access method uses less room on a disk and can be more efficient. With both methods you OPEN a file for input or output to access the device, and when you're finished you CLOSE the device. With CLOSE, just specify the file number, such as

```
CLOSE #2
```

The OPEN statement has two syntax forms:

```
OPEN mode,[#]f,fname [,file-buffer-size]
OPEN fname [FOR mode] AS [#]f [LEN=file-buffer-size]
```

The following example opens device #1 for output, "O", and calls the file TEST. We'll store four numbers in that file using WRITE *#f,vlist* and then close the file.

```
REM OPEN
A=5:B=-4:C=2.3:D=-6.4
OPEN "O",#1,"TEST"
WRITE #1,A,B,C,D
CLOSE #1
END
```

To read the data in, we open the device as input, "I". INPUT *#f,vlist* reads in the information, and PRINT prints the numbers on the screen.

```
REM READIN
OPEN "I",#1,"TEST"
INPUT #1,W,X,Y,Z
PRINT W;X;Y;Z
CLOSE #1
END
```

Using the other type of format for the OPEN statement and string variables for examples,

```
REM OPEN2
OPEN "TEST2" FOR OUTPUT AS #1
```

```
N$="BRETT LYNN"
S$="RICHARD LANE"
WRITE #1,S$,N$
CLOSE #1
END

REM READIN2
OPEN "TEST2" FOR INPUT AS #5
INPUT #5,A$,B$
PRINT A$
PRINT B$
CLOSE #5
END
```

You may also use PRINT *#f,vlist* to output information. With PRINT *#f,vlist* you do need to be careful that delimiters are between your items—such as a carriage return or a comma. You need to physically put in the commas between items, for example,

```
PRINT #3,A$,",",B$
PRINT#1, USING "###,";N1,N2,N3,N4
```

In the above examples we knew exactly how many items were in each file. This may not always be the case. You can put the statements in loops. EOF*(f)* tests for the end-of-file condition and returns −1 if there are no items left to read. LOC*(f)* returns an increment, or the number of bytes written to or read from the sequential file divided by the default size of 128 or the record size specified in the OPEN statement length.

In random disk files, LOC*(f)* returns the record number of the last record read or written. LOF*(f)* returns the length of the file in bytes.

The following sample program lets the user choose a filename to store the data and calls it L$. Then the user enters words or names by typing the name, pressing RETURN after each one. At the end RETURN is pressed without anything being entered. Up to 30 items may be entered. The program to get the names asks the user to enter the filename L$. This time a certain number of items are not known, so the information is read in a WHILE–WEND loop as long as the EOF is not −1.

```
REM SAMPLE I/O
T=0
PRINT "ENTER A FILENAME"
INPUT L$
```

```
PRINT "ENTER SEVERAL NAMES"
PRINT "PRESS RETURN AFTER EACH NAME"
PRINT "THEN TWICE WHEN YOU ARE FINISHED"
OPEN "O",#2,L$
ENTER:
 INPUT N$
 IF N$="" THEN FINISHED
 PRINT #2,N$
 T=T+1:IF T<30 THEN ENTER
FINISHED:
 CLOSE #2
 PRINT "FINISHED"
END
```

```
REM GETTING NAMES
DIM N$(30)
PRINT "ENTER FILENAME"
INPUT L$
PRINT:T=0
OPEN "I",#3,L$
WHILE NOT EOF(3)
 INPUT #3,N$(T)
 PRINT N$(T):T=T+1
WEND
CLOSE #3
END
```

To add to a sequential file, use "A" or APPEND for the mode. In the particular file the items will not replace the file, but will add at the end.

```
OPEN "A",#3,"TEST"
```

Random Files

To create a random access data file, use the mode "R".

```
OPEN "R",#4,"TESTDATA"
```

The random access files are stored in a packed binary format and thus take up less space on the disk. Also, to retrieve the information, you don't need to read sequentially through all the information. The information is stored in numbered records. The FIELD statement is used with random access files which allocate space for variables in a random file buffer. The format is

FIELD [#]*f*,*fieldwidth* AS *string*

where *f* is the file number and *fieldwidth* is the number of

characters to be allocated to the string variable. The FIELD command may consist of several *fieldwidth* AS *string* items.

```
FIELD #3,4 AS P$,16 AS F$
```

GET# reads a record from a random disk file into a random buffer and PUT# writes a record from a random buffer to the random access file. The formats are

```
GET [#]f[,record number]
PUT [#]f[,record number]
```

Examples are

```
FOR J=1 TO 30
 PUT #3,J
NEXT J
```

and

```
GET #2,X
```

When you save things in a random file, numeric values must be converted to strings. The procedure is to use MKI$*(e)*, MKL$*(e)*, MKS$*(e)*, or MKD$*(e)* to make a string out of the expression *e*, where I is integer, L is long integer, S is single-precision, and D is double-precision. Then use LSET or RSET to move the data from memory to a random file buffer. LSET left justifies the string in the field, and RSET right justifies the string. Since the string must be a certain length, spaces are used to pad the extra characters. Finally, write the buffer to the file using PUT#f.

To retrieve the information, keep in mind that numbers are in strings. CVI*(s)*, CVL*(s)*, CVS*(s)*, and CVD*(s)* are used with FIELD and GET statements to convert the strings back to numeric values. CVI converts a two-byte string to an integer. CVL converts a four-byte string to a long integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

Some examples are

```
INPUT ITEM$
LSET T$=ITEM$
INPUT N
LSET N$=MKI$(N)
PUT #3,C
...
GET #8,X
AB=CVI(G$)
PRINT USING "$$###.##";CVS(P$)
...
```

Program 2-1. Random File Example

```
REM RANDOM ACCESS
REM READ DATA
FOR I=1 TO 3
   READ M$(I)
   NEXT I
   DATA ADD NAME,RECALL NAMES,EXIT PROGRAM

 OPEN "R",#2,"MEALS",60
 FIELD #2,20 AS A$,20 AS B$,20 AS C$

REM TITLE SCREEN
INFO:
   WIDTH 60:CLS:LOCATE 10,26:PRINT "**** MEAL TICKET FILE ****"
   LOCATE 12,15:PRINT "STORE AND RETRIEVE YOUR BEST PALS' PREFERENCES"
   LOCATE 14,15:PRINT "FOR DINNER."
   LOCATE 16,15:PRINT "PRESS ANY KEY TO BEGIN."
GETKEY:
   R$="":WHILE R$="":R$=INKEY$:WEND
CHOICE:
   ON ERROR GOTO CHERR
   OPEN "PALS" FOR INPUT AS #3
   INPUT #3,N
   CLOSE #3
   X=N:CLS:LOCATE 3,17:PRINT " ************M E N U************"
   FOR Y=1 TO 3:LOCATE 4+Y,12:PRINT Y,M$(Y):NEXT Y
   LOCATE 10,12
   INPUT "YOUR CHOICE";R$
   IF R$ <"1" OR R$>"3" THEN CHOICE
   ON ASC(R$)-48 GOSUB ADD,REC,EX
```

```
   GOTO CHOICE
ADD:
   X=X+1
   CLS:INPUT "YOUR PAL'S NAME";NA$
   PRINT "HOW OLD IS ";NA$;:INPUT AG$
   PRINT "WHAT DOES ";NA$;" LIKE TO EAT";:INPUT FO$
   LSET A$=NA$
   LSET B$=AG$
   LSET C$=FO$
   PUT #2,X
   LOCATE 10,1:INPUT "ADD ANOTHER <Y/N>";R$
   IF R$="Y" OR R$="y" THEN GOTO ADD
   N=X
   OPEN "PALS" FOR OUTPUT AS #3
   WRITE #3,N
   CLOSE #3
   RETURN
REC:
   CLS:PRINT "(O)NE NAME OR (A)LL NAMES?"
GSTROKE:
   R$="":WHILE R$="":R$=INKEY$:WEND
   IF R$="O" OR R$="o" THEN GOTO ONENAME
   IF R$="A" OR R$="a" THEN ALLNAMES
   GOTO GSTROKE
ONENAME:
   CLS:PRINT "THERE ARE ";N;" NAMES ON FILE."
   INPUT "WHICH ONE";R
   IF R>N THEN ONENAME
   GET #2,R
   CLS:PRINT "PAL #";R;".....";A$:PRINT"AGE.....";B$
   PRINT "Serve to please.....";C$
```

```
    INPUT "ANOTHER NAME <Y/N>";R$
    IF R$="Y" OR R$="y" THEN GOTO ONENAME
    GOTO ENDREC
ALLNAMES:
    FOR Z=1 TO N
    CLS
    GET #2,Z
    PRINT "PAL #";Z;".....";A$:PRINT"AGE.....";B$
    PRINT "Serve to please.....";C$
    PRINT :PRINT :PRINT "PRESS ANY KEY TO CONTINUE"
WAITER:
    R$="":WHILE R$="":R$=INKEY$:WEND
    NEXT Z
ENDREC:
    RETURN
EX:
    CLOSE #2
    CLS
    END
NEWFILE:
    N=Ø
    OPEN "PALS" FOR OUTPUT AS #3
    WRITE #3,N
    CLOSE #3
    RETURN
CHERR:
    IF (ERR<>53) THEN EX
    GOSUB NEWFILE:RESUME CHOICE
```

Chapter 3

Getting Started with AmigaDOS

Charles Brannon

Getting Started with AmigaDOS

Charles Brannon

The Commodore Amiga comes with a large looseleaf binder packed with information on this advanced computer. Even if you've never used a graphics-oriented operating system before, you can plug in the mouse and be up and running on the Amiga Workbench in very little time.

But there's something missing from the standard manuals: instructions for using AmigaDOS, a powerful alternative to the Workbench. Although the Workbench is a versatile tool for both beginners and expert users, there are also advantages to a command-driven operating system. With AmigaDOS, you can gain finer control over the computer and its many functions—at the expense of having to memorize dozens of commands and their proper syntax. These tradeoffs have been a subject of hot debate ever since the Macintosh made its debut three years ago. Fortunately, the Amiga gives you both options. And thanks to its multitasking capabilities, you can even flip back and forth between both systems at will.

All this is made possible by the Amiga's multilevel operating system. The core is Intuition, a package of efficient subroutines designed to ease the software designer's task. It's filled with routines needed by almost every program, saving programmers the trouble of reinventing the wheel. Intuition includes powerful graphics utilities so programmers needn't program the computer at the hardware level.

Pay No Attention to the Little Man

Attached to the Intuition core is AmigaDOS, which itself has two levels. First, AmigaDOS provides all the disk operating system functions for the computer, such as managing, opening, accessing, updating, and closing files; buffering direct memory access (DMA) for the disk drives; supporting named devices; and allocating memory.

Second, AmigaDOS as a *tool* provides one or more Command Line Interfaces (CLIs). A CLI is a traditional command-oriented operating system interface, much like CP/M, MS-DOS, and PC-DOS—but even more powerful. At a screen

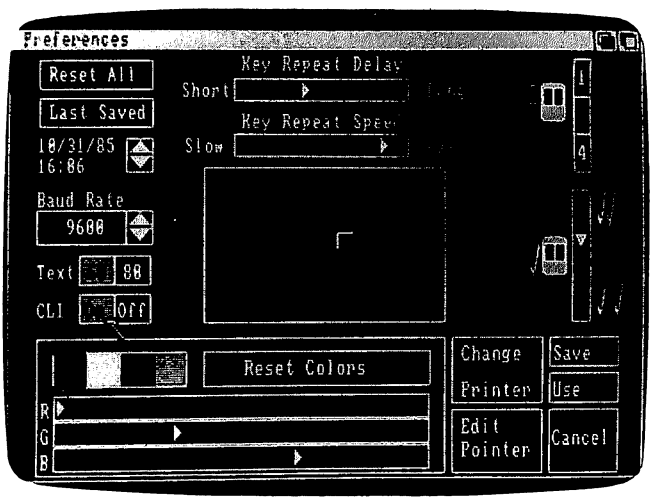
prompt, you can type in commands to load and run programs; list disk directories; copy, rename, and delete files; and even write simple programs called batch files.

When you start the Workbench, AmigaDOS comes with it. In fact, you've undoubtedly seen the AmigaDOS screen briefly appear when you first boot up the Workbench disk. AmigaDOS comes up first, loads the Workbench, then shuts down its CLI, transferring control to the Workbench.

AmigaDOS is like the Wizard of Oz. It pulls the strings of the marionette that is the Workbench. Meanwhile, hidden from sight, AmigaDOS is doing much of the work. When you step behind the curtain, you see how things are really done. Once the object-oriented illusion of the Workbench is stripped away, you find yourself working with files, streams, subdirectories, and pathnames.

Starting a CLI

To start an AmigaDOS CLI, first run the Preferences tool by opening up the Workbench disk and double-clicking on the Preferences icon. The Preferences screen (see photograph) has an option box labeled CLI [ON] [OFF]. Click the box ON, then click on SAVE. The Workbench file on the disk will be updated, and now the CLI option will be available whenever you start the Workbench in the future.

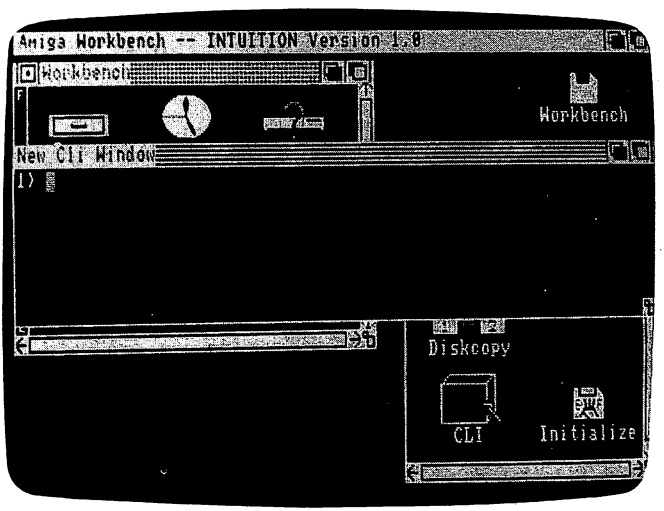


To allow access to AmigaDOS from the Workbench, click the mouse button with the pointer positioned upon the CLI [ON] box within the Preferences screen.

With CLI enabled, open the Workbench's System folder. In addition to the usual icons for Disk Copy, Icon Editor, and Initialize, you'll see a cube-shaped icon marked with 1> and labeled CLI. Double-click on this icon to open a CLI window.

The first thing you'll notice in the window is the 1> prompt. Unlike DOS prompts on most other computers, this doesn't represent the current disk drive. Instead, it represents the *task number* assigned to the window. AmigaDOS is one of the few microcomputer operating systems that can multitask itself.

To see how this works, enter NEWCLI at the 1> prompt. When you press RETURN, a second CLI window pops up with the prompt 2>. This CLI is a complete, full-powered CLI, independent from the first CLI. In effect, you now have two command-driven operating systems running on the computer. Each window can execute a different DOS task. While one CLI is busy printing a file, you can go to another CLI window to list a directory.



Clicking on the CLI icon from the Workbench opens up this AmigaDOS screen window.

Although several CLI windows can display output simultaneously, only one CLI window at a time can accept input. To select which CLI is *active*, point to its window and click the mouse button. You can distinguish active from inactive win-

dows by glancing at the title bars—the bar of an inactive window is dimmed.

If you type NEWCLI at the 1> or 2> prompt, a third CLI window opens with a 3> prompt. How many CLI windows can be opened at once? On a 512K Amiga, we've opened as many as 20 CLIs before encountering an out-of-memory message.

When you're done with a CLI, close it by entering ENDCLI. When you close the primary CLI, control reverts to the Workbench.

AmigaDOS Devices

For any DOS commands to work, the startup (Workbench) disk must be in the current drive. Unlike other operating systems, AmigaDOS contains no memory-resident commands. All commands are *extrinsic*—they're loaded from disk only when called. AmigaDOS always looks for commands first from the current directory, then the C subdirectory on the SYS: (startup) disk. We'll elaborate on this in a moment.

You can type AmigaDOS commands and filenames in either upper- or lowercase (for clarity, all our examples are shown in uppercase). If you make any typing mistakes, you can press BACK SPACE or cursor left to retype. Type CTRL-X to erase the whole line. You can get a complete list of all commands by typing DIR SYS:C. This shows the contents of the C subdirectory on the startup disk, the directory where all AmigaDOS commands are stored.

The DIR command displays the current directory. By default, the current directory is listed from the internal drive, which is referred to as DF0:. If you have a multiple-drive system, you can get a directory of the first external drive by typing DIR DF1:. Up to three external drives can be daisychained, numbered from DF1: through DF3:. The colon following the drive name is important—it tells AmigaDOS that it is a *device name* rather than the name of a file.

A special device, SYS:, refers to the system (startup) disk. Although the startup disk is usually in drive DF0:, SYS: is not necessarily synonymous with DF0:. SYS: refers to the startup *disk*, not a *drive*.

Disk Names

Instead of referring to a physical drive, you can access a disk by name. When you use Workbench to copy or format a new disk, the disk is assigned a unique name, which is displayed beneath the disk icon on the Workbench screen. When specifying a disk name in a command, you must end it with a colon, as you do with device names. If the disk is not in a drive when you refer to it in a command, AmigaDOS prompts you to insert it.

The ability to specify disk names is vital with single-drive Amigas. When you type DIR, the DIR program is loaded from the Workbench disk and displays the directory of that disk. If you insert another disk and type DIR, you have to reinsert the Workbench disk so that AmigaDOS can read the DIR file. Unfortunately, AmigaDOS doesn't ask you to put the other disk back in—so you still get the directory of the Workbench disk.

The solution? Follow the DIR command with the proper disk name. For example, DIR "BASIC Demos:" (remember the colon) calls a directory of the disk named BASIC Demos. AmigaDOS still loads the DIR command file from the Workbench disk, but now asks you to insert "BASIC Demos" before displaying the directory. Specifying the disk name (also known as a *volume name*) forces AmigaDOS to refer to a *disk* instead of a *drive.*

Other device names are PAR: for the parallel printer port, SER: for the serial/modem port, PRT: for whatever printer port you've specified via the Preferences tool, and RAM: for the RAM disk. Another device, NIL:, is a null handler. It accepts output instantly, but does nothing with it. The NIL: device is useful for testing a program without wasting paper or time—just redirect the output to NIL:.

The RAM disk behaves just like a very fast disk drive except that its contents are lost when the computer is rebooted or turned off. Be sure to copy anything important from the RAM disk to a real disk before shutting down, or even more frequently if power failures and brownouts are common in your area. The RAM disk is dynamic: Unlike some RAM disks, it has no fixed size. It starts out empty, then grows or shrinks as you add or remove files. Therefore, it's always 100 percent full, using only as much memory as it needs to hold the files you've stored there.

Whenever you want to refer to the RAM disk in an AmigaDOS command, just precede a filename with the prefix RAM:. At present, the RAM disk isn't accessible from the Workbench.

Another special device name, * (the asterisk), refers to the current keyboard/screen device. Input from * is from the keyboard; output to * appears in the current window. Notice that this is different from the use of * as a wildcard character in some other operating systems.

Understanding Pathnames

A *file* is the basic data storage object in AmigaDOS. A file is addressed by a *filename*, a string of up to 30 characters. Each file must have a unique filename. Filenames can include almost any character, including characters such as space, =, +, and ", special AmigaDOS delimiters that you should avoid. (If a file contains special characters, you can enclose them in quotation marks to make sure the special characters aren't acted upon by AmigaDOS.) However, two characters are forbidden in filenames by AmigaDOS—the colon (:) and the slash (/).

Each drive has its own *directory*, a list of all filenames and subdirectory names. A *subdirectory* is a directory within a directory. Subdirectories are like drawers on the Workbench. You can even nest subdirectories within subdirectories within subdirectories, which can get confusing.

You separate a subdirectory name from a filename with the slash (/). Notice that this slash leans in the opposite direction from the backslash (\ ) used in IBM PC-DOS for subdirectories.

A complete filename can be as simple as PROCEDURES, equivalent to DF0:PROCEDURES, since DF0: is the default drive. Filenames can also be a lot more complicated, for example, DF1:BASIC PROGRAMS/GIDGET, which refers to the program GIDGET in the subdirectory BASIC PROGRAMS on the external drive, or, another example, RAM:LOGO/DEMOS/SPINNER, which refers to the file SPINNER in the DEMOS subdirectory which is in the LOGO subdirectory in the RAM disk.

Fortunately, there are shortcuts. Instead of entering the current pathname, such as DF0:DEMOS/DOTS.INFO, it's sufficient to use DOTS.INFO if the current directory is DF0:DEMOS. We'll show below how to change the current directory.

More About Multitasking

You can do nearly everything with AmigaDOS that you can with the Workbench. There are commands to copy files, delete files, rename files, format disks, send listings to printers, set date and time, and more. You can also run any application program from AmigaDOS.

All Workbench programs have two files: one file that contains the program and another file with an extension of *.INFO* that contains icon information for the program. For instance, the icon for the Preferences tool is drawn from PREFERENCES.INFO. To run the Preferences tool from AmigaDOS, enter PREFERENCES at a CLI prompt. Similarly, enter CLOCK to start the clock tool.

Be careful not to let the program you're running override the CLI. If you'd like to keep the CLI going while running another program, preface the AmigaDOS command with another command, RUN. This starts a new, simultaneous program. RUN CLOCK starts the clock while permitting the CLI to continue running. The clock becomes a new CLI task. We've used this feature on a 512K Amiga to run MetaComCo ABasiC simultaneously with AmigaDOS, the Workbench, and a full-screen editor.

A Custom DOS Disk

It's fairly simple to create an AmigaDOS-only disk. This disk can be used whenever the system asks for a Workbench disk. You probably won't want to modify your original Workbench disk, however. It's better to modify a copy of it and set aside the original for safekeeping. You can make several copies of your AmigaDOS disk for future use, if you want. Just follow these steps:

1. Open the System drawer on the Workbench disk. If you don't see the CLI icon—a small cube labeled with a 1> symbol—run Preferences (otherwise, continue to step 2). One of the settings on the first Preferences screen is labeled CLI [ON] [OFF]. Click it ON, then click on the Save box to save the change to disk. Return to the Workbench and reopen the System folder. You should now see the CLI icon.
2. Double-click on the CLI icon. A window titled "New CLI Window" appears. Click inside the window to make the CLI active.

3. At the 1> prompt, type ED S/Startup-Sequence and press RETURN. This loads a program called ED, a full-screen editor, and loads the file Startup-Sequence from the S subdirectory. Startup-Sequence is the batch file that makes AmigaDOS automatically start the Workbench when you boot the Workbench disk. After ED starts, you should see something like this on the screen:

```
ECHO "WorkBench Disk. Release 1.1"
ECHO " "
ECHO "Use Preferences tool to set date"
ECHO " "
LoadWb
endcli > nil:
```

   These are the batch file commands that AmigaDOS executes each time you boot up the Workbench disk. The ECHO commands are similar to PRINT statements in BASIC; they merely display messages on the screen. The last two commands in this file are the ones we're interested in changing.
4. Using the cursor keys, move the cursor to the line with the LoadWb command and press CTRL-B twice to erase the last two lines. The batch file should now consist of the four ECHO commands only. If you wish, you can change the text in the ECHO commands to give your boot disk that "personal touch."
5. Press the ESC key. An asterisk prompt (*) appears at the bottom of the screen. Type X at this prompt and press RETURN. This exits the ED program and saves the new Startup-Sequence file to disk. If you've made a mistake and would like to start over, press ESC-Q to quit the editor without changing the file.
6. After the disk busy light goes off, simultaneously press CTRL and both Amiga keys on each side of the space bar to reboot the system. This time, and from now on whenever you boot with this disk, AmigaDOS ends up in memory instead of the Workbench.

The Workbench Option

To conserve space on your new AmigaDOS disk, you may want to erase some files used by the Workbench, such as the LOADWB command in the C subdirectory, the Notepad, the clock, and all .INFO files. However, it's convenient to have the

Workbench available when you need it. You could use the editor to create another batch file that includes LOADWB and ENDCLI > NIL:. You would then type EXECUTE WB at a CLI prompt to bring up the Workbench (assuming you named the batch file WB by typing ED WB to create the batch file). ED is useful for creating all kinds of simple batch files, in fact.

AmigaDOS Commands

Following is a list of the most useful AmigaDOS commands with brief descriptions and examples. Some commands shown here may not be available on your copy of AmigaDOS/Workbench, while there may be other commands available to you that have not been documented. This chapter was prepared with AmigaDOS version 1.0 and 1.1. Type DIR SYS:C at a CLI prompt to see a complete list of available commands. When you're experimenting with AmigaDOS commands, we strongly recommend that you use a scratch disk to avoid wiping out an important file or even a whole disk. (A complete list all AmigaDOS commands for versions 1.0 and 1.1 appears in Appendix B.)

< and > (Input/output redirection). These symbols redirect the normal input/output flow of a command. For example, a program that normally accepts input from the keyboard and prints its output on the screen could be coerced into accepting input from a file or to send its output to the printer. The < and > symbols are used to point in the direction that I/O should flow; the less-than sign (<) redirects input, and the greater-than sign (>) redirects output. When using < to redirect input, you may need to use a question mark for the parameter that the redirection file is replacing.

Examples:

DIR > DIRFILE

This redirection of the DIR command sends the disk directory to the file DIRFILE instead of to the screen. To confirm this, you can enter TYPE DIRFILE to display the contents of DIRFILE.

STACK < BASIC.STACK ?

The stack command normally accepts a command line parameter. Here, a file (BASIC.STACK) containing the number 8000 can be substituted. In order for the file to replace the

command line parameter, you must use a question mark to hold that parameter's position.

CD *(Change Directory).* Follow CD with the pathname of the directory you'd like to work with. Entering CD by itself displays the current search path. When you type a command, AmigaDOS first searches for the extrinsic command file in your current directory, then in the COMDIR directory. AmigaDOS also looks for all filenames in the current directory unless you override the current directory with another pathname.

Example:

CD DF1:BASIC

This switches the current directory to the first external drive and the subdirectory BASIC.

COPY. This copies a file or group of files to any legal destination. The keyword TO specifies the destination path. You can use the optional keyword FROM to specify a directory other than the current directory. If you are copying entire subdirectories, append the keyword ALL so that COPY creates a subdirectory in the destination directory. COPY normally displays the name of each file as it's copied. Append the keyword QUIET if you'd like to suppress this.

Examples:

COPY MATRIX.SORT TO DF1:MATRIX.BKP

This copies the file MATRIX.SORT in the current directory, creating a file called MATRIX.BKP in the main directory of the first external drive.

COPY FROM DF1:GOBBLE TO DF0:

This copies the file GOBBLE from the external drive to the internal drive.

COPY DF0: TO DF1: ALL

This backs up the entire contents of the internal drive onto the external drive, including the contents of all subdirectories. COPY doesn't format the destination disk, so DISKCOPY is a more convenient way of backing up an entire disk.

COPY SYS:C TO RAM: QUIET

This copies the command directory to the RAM disk without listing all the filenames.

COPY * TO PRT:

This accepts lines from the keyboard and prints them on the printer until CTRL- \ is pressed.

DATE. This command sets the current date and time. When you create or update a file, AmigaDOS stamps the date and time on the directory. Since there's no battery-backup for the clock, however, the Amiga doesn't know this information until you tell it. By default, AmigaDOS assumes the date stamped on the most recent file. Entering DATE by itself displays the current date.

To set the date from AmigaDOS without running the Preferences tool, follow the DATE command with a date in the form *DD-MMM-YY* (for example, 25-DEC-85). To set the time, follow this with the form *HH:MM* (using 24-hour time, such as 13:00 for 1:00 p.m.). You can type DATE TOMORROW to advance the date ahead one day, or DATE YESTERDAY to back up one day. Another shortcut is simply to enter DATE *dayname,* as in DATE TUESDAY. If you use your Amiga frequently, this may be all you need to keep things up-to-date.

An interesting application of the DATE command is to determine which day of the week a certain date falls on. For example, DATE 25-DEC-86 sets the date to Christmas Day, 1986. If you then enter DATE by itself, AmigaDOS displays THURSDAY 25-DEC-86, letting you know that Christmas falls on a Thursday in 1986.

Examples:

DATE 04-JUL-76

This sets the current date to July 4, 1976. (The Amiga assumes you know which century you're living in, so there's no way to specify 1776 versus 1976 or 2076.)

DATE 08:30 FRIDAY

This sets the time to 8:30 a.m. and advances the date to Friday. DATE FRIDAY 08:30 would also work.

DELETE. This command deletes a file or group of files. Follow DELETE with the pathname specifying a file. You cannot delete a subdirectory if it contains any files. You can delete several files by separating each one with a comma, up to a maximum of ten. DELETE doesn't ask ARE YOU SURE?, so be careful.

Examples:

DELETE MASTER.BKP

This deletes the file MASTER.BKP from the current directory.

DELETE DF1:PROGS/ALPHA,OMEGA

This deletes the file ALPHA on the PROGS subdirectory on the external drive, and also deletes the file OMEGA from the current directory.

DIR *(Directory)*. DIR and LIST are similar commands. DIR lists just file and directory names, while LIST gives additional information (see LIST). Follow DIR by a legal directory path. Don't include the name of a file in the path. The OPT command permits special directory options. DIR OPT A lists the contents of any subdirectories along with the main directory. DIR OPT D lists only subdirectory names.

There is a special interactive directory mode which you enter with DIR OPT I. While in directory mode, the entries are displayed one at a time. Press RETURN to go on to the next entry. If the entry is a subdirectory name, you can press E to enter that subdirectory, listing its files. To exit a subdirectory, enter B. If the current entry is a file, you can enter T to type its contents (CTRL-C aborts the display). You can enter the command DEL to delete the current entry (again, you can't delete a directory unless it's empty). Type Q to quit the interactive mode.

Examples:

DIR

This displays the current directory.

DIR DF1:DEMOS

This displays the contents of subdirectory DEMOS on the external drive.

DIR DF1: OPT A

This displays the directory and the directory of next-level subdirectories on the external drive.

DISKCOPY. To copy one disk to another with two drives, enter DISKCOPY DF0: TO DF1:. Formatting is automatic, and the copy has the same name as the original unless you use the NAME option, as in DISKCOPY DF0: TO DF1: NAME "KICKSTART BACKUP". To copy a disk with one

drive, type DISKCOPY DF0: TO DF0:. You'll be prompted to insert the original and destination disks alternately.

Examples:

DISKCOPY DF1: TO DF0:

This backs up the disk in the external drive to the disk in the internal drive. Although both disks will have the same name, AmigaDOS can distinguish between them by the dates they were created.

DISKCOPY DF0: TO DF0: NAME "WORKBENCH BACKUP"

This creates a named backup of the disk in the internal drive. Several disk swaps are required.

ENDCLI. This cancels the current CLI window. Use this command only to terminate a secondary CLI or to return to the Workbench. If there is no Workbench and you close the primary CLI, everything ends, leaving you nothing to work with. Your only recourse would be to reboot the system.

FILENOTE. This command attaches a comment to a file. Although AmigaDOS's 30-character filenames let you be quite descriptive, an optional FILENOTE lets you attach an additional 80-character comment to a file. This comment is displayed beneath the filename when you use the LIST (not DIR) command. Follow FILENOTE with the name of the file you're describing, then the comment. You must enclose the comment in quotation marks if it includes spaces. The FILENOTE command also lets you include two optional keywords, FILE and COMMENT, presumably for the sake of readability.

Files have no comment by default. The comment is retained if the file is changed or overwritten. However, if you copy a file, its filenote does not get copied with it.

Examples:

FILENOTE waver.bas "Program lets you create sound waves."

After you attach this comment to the file waver.bas, LIST waver.bas yields this result:

waver.bas 2272 rwed 11-Oct-85 10:09:53
: Program lets you create sound waves

Second example:

FILENOTE FILE waver.bas COMMENT "Program lets you create sound waves."

This is identical to the first example, except for the optional keywords FILE and COMMENT.

FORMAT. This lets you format a new disk. Follow FORMAT with the keyword DRIVE (required), a drive device, the keyword NAME, and a unique 30-character disk name (enclosed in quotation marks if it contains any spaces). FORMAT customizes a blank disk for use with the Amiga drives. Don't forget that FORMAT irreversibly erases everything on the disk.

Example:

FORMAT DRIVE DF0: NAME "FINAL PROTOTYPE"

INFO. This command shows a disk report. INFO displays the size of each mounted drive (normally 880K, except for the RAM disk), the number of sectors used, number of sectors free, percentage of capacity used, number of disk errors that have occurred, the read/write status, and the disk's name. INFO also separately displays the names of the currently inserted disks. INFO has no additional parameters. Use LIST to display information about a particular file or directory.

INSTALL. This command makes a disk bootable. In other words, an INSTALLed disk can be inserted at the Workbench prompt to bring up the system. Just follow INSTALL with the optional keyword DRIVE and the drive number. If you want to be able to execute AmigaDOS commands after booting, you must copy the C subdirectory from your master disk onto the copy. (All AmigaDOS commands are extrinsic and contained in the C subdirectory.)

Example:

INSTALL DRIVE DF1:

This makes the disk currently mounted in the external drive bootable.

JOIN. This command combines two or more files. Follow JOIN with up to ten filenames separated by spaces. The destination file, holding the conglomerate, is specified with the keyword AS. The original files are unchanged.

Example:

JOIN Checks/Oct Checks/Nov Checks/Dec AS "Checks/4th Quarter"

This combines the files Oct, Nov, and Dec from the subdirectory Checks into a single file called "4th Quarter" to

be created in the Checks subdirectory. The destination filename is enclosed in quotation marks because it contains a space character.

LIST. This command gets you more information about a disk, directory, or file. LIST by itself displays the current directory. LIST can also be followed by a directory path and/or a filename. LIST followed by a filename gives information only for that file. For each file, LIST displays the filename, size in bytes, file access (Readable/Writeable/Executable/Deletable), the date stamp, and the comment, if one was specified with the FILENOTE command (FILENOTE uses the form FILENOTE *filename "comment"*).

LIST can also be used with the keyword TO, which can redirect the listing to another device, such as the printer. With DATES, LIST displays dates as *DD-MMM-YY*, which is the default unless you use NODATES. You can use SINCE followed by a date to show only those files written on or after the specified date, or UPTO to list only those files created before or on the specified date. (The date follows the same format used by the DATE command.)

Example:

LIST DF1: SINCE YESTERDAY

This displays the main directory of the external drive, including only those files which were created yesterday or today.

MAKEDIR *(Make directory)*. Follow MAKEDIR with a new directory path. The last directory name in the path is the name of the new directory.

Examples:

MAKEDIR "AIR MAIL"

This creates a new subdirectory called "AIR MAIL" (quotation marks used because name contains spaces) on the current directory.

MAKEDIR DF1:DEMOS/GRAPHICS

This creates a new subdirectory called GRAPHICS within the existing subdirectory DEMOS on the disk in the external drive.

NEWCLI. By itself, NEWCLI just opens up a new CLI window and transfers keyboard control to it. The original CLI

is retained. You can use the mouse to move and resize the window as usual. This new CLI can use settings different from other CLIs, such as a unique current directory. A CLI can work in the background while you switch to another process. You can customize a CLI by following it with "CON: *x/y/width/height/title*", which lets you specify the starting position, size, and name of the new CLI window.

Although not documented, it's possible to control a CLI with another device. NEWCLI SER:, for example, starts a CLI controlled by an RS-232 device, such as a modem or terminal. This could let a remote user control his or her own independent DOS console.

Use ENDCLI to cancel a CLI and revert to a former one.

Example:

NEWCLI "CON:320/100/160/50/ EXTERNAL DRIVE"

This creates a 160 × 50-pixel window at position (320,100) with the name "EXTERNAL DRIVE". This new window is a complete CLI. With the CD command, you can set up this window to access one drive and a different window to access another. The parameters of the CON: device, shown here, can be used as the output of other commands as well.

PROMPT. Defines a new CLI prompt. Follow PROMPT with a message, enclosing it in quotation marks if the message contains any spaces. The message is a replacement for the normal 1> or 2> prompt of AmigaDOS. You can embed the characters %N to display the current task number.

Examples:

PROMPT "%N> "

Displays the default prompt.

PROMPT "Ready, Master:"

Displays Ready, Master: as the new AmigaDOS prompt.

PROTECT. This command sets a file's protection status. Follow PROTECT by the filename, the optional keyword STATUS, and the protection desired: **r** to allow a file to be read, **w** to allow a file to be written to, **d** to make a file deletable, and **e** to make the file executable. To protect a file against a certain type of access, omit the corresponding letter. Only actual machine-runnable object code programs should be made executable. In versions 1.0 and 1.1, only the delete status is in effect;

files cannot be protected from being written to, read, or executed.

Examples:

PROTECT YUPPIES

This makes the file YUPPIES practically nonexistent. It shows up on the directory, but it cannot be read, written to, deleted, or executed. You can use PROTECT again to override this, of course.

PROTECT "DON'T READ ME" STATUS WD

This allows the file "DON'T READ ME" to be written to and deleted, but not read or executed. PROTECT provides a simple form of protection since it can always be used to change the file's status back. It mainly protects you against your own mistakes.

RENAME. Follow RENAME with the optional keyword FROM, the existing name of the file, the optional keyword TO or AS, and the name you'd like to change it to. The new name must not conflict with any existing name. The position occupied by that file on the directory may change after the rename, especially if you use a different subdirectory name for the new name.

Examples:

RENAME FROM "Templates/Amortization" TO "Templates/32yr Amortz"

This changes the name of file Amortization to "32yr Amortz" within the subdirectory Templates.

RENAME Dog AS Cat

This changes file Dog to Cat within the current directory.

RENAME FROM Progs/Slither TO Pascal/Slither

By changing Slither's subdirectory name, we have, in effect, moved Slither from the Progs directory to the Pascal directory. (This is similar to the usage of mv in the Unix operating system.)

RUN. This lets you run any executable file "in the background," that is, while another task is running. RUN is the AmigaDOS multitasking command. If you start an object module or command by just typing its name, it takes over control from AmigaDOS. Some commands don't return to AmigaDOS

when they end, locking you out of the CLI. RUN lets you run any command or program as an independent, simultaneous process, just as NEWCLI creates a simultaneous CLI. You can run multiple commands and programs by ending each line with a plus sign (+) to specify a continuation to the next line.

Example:

RUN ED Simple

This starts the full-screen editor with the file Simple. Meanwhile, the CLI is still running. To get to it, use the mouse to select the current screen's back gadget to display AmigaDOS, then click in the AmigaDOS window to activate the CLI. You can type in the AmigaDOS window, executing commands, then switch to ED to continue editing. Without RUN, ED takes over until you exit.

SAY. SAY is used to invoke the Amiga's built-in speech synthesis capabilities. The user can control the quality and speed of speech. SAY has two modes: interactive and direct.

In direct mode the text to be spoken or an AmigaDOS file containing the text to be spoken is specified on the command line with SAY.

Interactive mode is entered by typing SAY by itself. Two windows will appear on the system screen.

The "phoneme" window displays the option codes that may be used to control the quality and speed of the synthesized voice.

The "Input" window is where text that you wish spoken is displayed as it is typed on the system keyboard. The text is passed to SAY when the RETURN key is pressed. Interactive is exited by typing a line consisting only of a RETURN keystroke.

The SAY command was added to AmigaDOS in version 1.1.

SAY *[options] [text]*.....

[options]	Control the quality, pitch, speed, and source of the text to be spoken. SAY identifies options by a leading dash (–). Valid options for SAY are
–f	Use female voice.
–m	Use male voice.
–n	Use natural voice.
–r	Use robot voice (monotone).
–p###	Set pitch of voice to ### (valid values are 65–320).
–s###	Set speech rate to ### (valid values are 40–400).

-x <file> Say contents of <file>. The -x option may not be invoked in the interactive mode of SAY; <file> must be an AmigaDOS file in the current directory and may not contain any spaces or be enclosed in parentheses.

Multiple options may be specified at one time.

Example:

SAY -f -p250 -s130 The Amiga can talk like a female

SEARCH. Finds text within files. This command searches for the target string through any directories you specify. Follow SEARCH with the optional keyword FROM, the pathname of the directories to be searched, the optional keyword SEARCH followed by the search string, and the optional keyword ALL, which forces SEARCH to look through all subdirectories contained in the specified directory. When SEARCH finds the target string, it displays the line containing the string as well as the line number of the line containing the string. If you're searching through a directory, SEARCH also displays the filename of each file it's searching through.

SEARCH is not case-sensitive; it matches regardless of upper- or lowercase. You can cancel the command with CTRL-C. To force SEARCH to abandon the current file and begin searching the next, press CTRL-D. During a search, you may see the message "Line *xx* truncated." This isn't anything to worry about; it just indicates that the line was too long to be searched, so if your search string was contained somewhere near the end of a too-long line, the search program could not find it.

Examples:

SEARCH FROM DF0: SEARCH LoadWb ALL

This looks for the phrase LoadWb. The entire contents of the internal drive are searched, including all subdirectories, so this command takes a long time to finish.

SEARCH Progs/Tempfile LIBRARY

This looks for the word LIBRARY in the file Tempfile within the subdirectory Progs.

SORT. This command alphabetically sorts a file you specify. Each record in the file to be sorted must end with a carriage return. Use SORT followed by the optional keyword FROM, the file to be sorted, the optional keyword TO, and the name of the file where the sorted output should be stored.

SORT collates based on the entire line unless you include the keyword COLSTART and a column number. The sort comparison then starts by comparing two lines from that column to the end of the line. If that partial comparison succeeds, the first portion of the line is compared. This lets you specify two levels of sorting (see example).

Unless the file to be sorted is less than about 200 lines, increase the stack size with STACK to prevent a crash (see below). It's better to use too much stack space than too little.

Example:

If you have a list of first and last names, with the first name and initial in columns 1–19, and the last name always starting in column 20, you could use

SORT FROM Route TO Sorted.Route COLSTART 20

The files are sorted by last name, and each group of identical last names is subsorted by first name.

STACK. Sets the stack size. Follow STACK with the new stack size in bytes. The normal stack size is 4000, sufficient for most commands. When using SORT, MetaComCo ABasiC, programs with lots of nested subroutines, or programs using flood-fill, you may need to increase the stack size to prevent a crash. A value from 8000 to 10,000 is usually generous enough for these cases.

TYPE. This command prints out a file on the screen. It's generally used with text files. Displaying other types of files usually produces nonsensical streams of strange characters. Follow TYPE with the filename. To redirect TYPE to another device, include the TO option, as in TYPE README.DOC TO PRT:.

TYPE allows two options. TYPE OPT N creates sequential line numbers for each line of text. You could use TYPE SAMPLE TO "NUMBERED SAMPLE" OPT N to create a line-numbered version of SAMPLE as "NUMBERED SAMPLE". TYPE OPT H displays the characters in a file as hexadecimal numbers. This is more useful when displaying machine language code or data files.

Examples:

TYPE "DF1:BASIC PROGRAMS/PINPOINT"

This displays the BASIC program PINPOINT located in the subdirectory BASIC PROGRAMS in the external drive. In

this case, quotation marks are required to prevent the embedded space in BASIC PROGRAMS from terminating the TYPE command.

TYPE SYS:C/DIR OPT H

This displays the contents of the DIR command (which is stored as a file in SYS:C) in hexadecimal. (Unless you can mentally disassemble the hex dump into 68000 mnemonics, this file will make no sense.)

WAIT. This makes AmigaDOS pause and do nothing for a span of time. Although this might seem a dumb command, WAIT has certain advantages over walking away from the computer or simply turning the machine off. Only the current CLI is frozen; multitasked processes continue. WAIT by itself pauses for one second; you can follow WAIT with a number of seconds, followed by either SEC or SECS, and a number of minutes, followed by either MIN or MINS. You can optionally include the keyword UNTIL followed by a time of day, specified as *HH:MM* (as measured by the Amiga's internal clock, so make sure it's set correctly). WAIT is useful within batch files to allow time for a message to be read or as a background task to wait until a particular time before executing another command.

Examples:

WAIT 10 MINS 20 SECS

Waits for 10 minutes, 20 seconds.

WAIT UNTIL 17:00

Waits until the current time is 5:00 p.m.

```
RUN WAIT 10 SECS +
DIR +
ECHO "All done."
```

Waits for 10 seconds, calls a directory as a second CLI task, then prints the message "All done."

WHY. This interesting command calls up an additional explanation of what caused the most recent error. When an AmigaDOS command fails, you'll usually get a terse error message. If you want a more detailed, technical description, ask WHY. However, many times WHY isn't any more helpful—it just explains in more detail why a command failed.

Example:

WAIT 10 SECONDS

AmigaDOS responds with the error message "Bad Args" because the correct notation is WAIT 10 SECS, not WAIT 10 SECONDS. If you type WHY, you get this answer:

Last command failed because argument line invalid or too long.

Although more descriptive, it still doesn't explain that SECONDS should be SECS—but it does point you in the right direction.

Chapter 4

AmigaDOS Batch Files

Charles Brannon

AmigaDOS Batch Files

Charles Brannon

AmigaDOS is more than a console-driven disk operating system. By executing a sequence of AmigaDOS commands stored in a file, AmigaDOS takes on some of the characteristics of a programming language. Whether you want to simplify repetitive disk commands or create personalized custom commands, batch files further extend the range and flexibility of AmigaDOS.

No matter how easy it is to use a program, the most popular programs are those that give users more power. And although a program may have scads of powerful commands, the most powerful programs are those which let users put the commands together in new ways—in effect, to write programs.

Instead of forcing you to always issue commands one at a time, a programmable application lets you create a script of commands to customize the behavior of the program. Whether we're talking about word processing macros, spreadsheet templates, relational database languages, or advanced machine language, programmability is the real key to software power. If you feel limited by a certain range of commands, you can combine the commands in new ways to create personalized features, just as we combine the vocabulary of English words to create a wealth of literature. Why just read when you can write?

Scripts, Sequences, and Batches

AmigaDOS is more than just a disk operating system—it's a programmable system that can process lists of its own commands as well as individual commands. In effect, AmigaDOS is a simple disk-oriented programming language.

A list of AmigaDOS commands can be stored in a disk file variously known as a script, a sequence, or a batch file. The term *batch file* is most commonly used by those who work with PC-DOS, MS-DOS, and CP/M, which are also programmable disk operating systems. To keep things straight, we'll use *batch files* synonymously with *scripts* and *sequences.*

Even if you don't program in BASIC or any other language, you may be interested in learning about AmigaDOS batch files. The batch file "language" is simply made up of the

same AmigaDOS commands you've probably been using all along (see Chapter 3 and Appendix B). There are also a few AmigaDOS commands designed especially for batch files.

Creating and running batch files is easy. Using a text editor, you just type in a list of AmigaDOS commands. Then you save the list on disk under a filename. To run the batch file, you type EXECUTE *filename* at an AmigaDOS prompt. AmigaDOS reads the batch file and executes the list of commands, just as if you had typed them one by one yourself.

We won't cover some of the more advanced features of batch files, useful only to advanced C and machine language programmers. Instead, we'll concentrate on the everyday utility of batch file programming.

A Quick Example

In a moment, we'll show how to create batch files with ED, the AmigaDOS full-screen text editor, but, first, there's a simpler way to create a short batch file. Enter this line at an AmigaDOS prompt:

copy * to Hello

(Note that AmigaDOS commands can be entered in uppercase or lowercase.)

Although nothing seems to happen, AmigaDOS is waiting for you to enter some lines. We'll use the ECHO command to display a friendly message. ECHO displays any text that follows it within quotation marks, just like the PRINT statement in BASIC. One difference is that if you want to ECHO only a single word, the quotation marks aren't necessary.

At an AmigaDOS prompt, enter the following text, pressing RETURN after each line:

```
echo "Hello!"
echo "I am your friend, the Amiga"
echo "personal computer."
```

After the last line, press CTRL- \ (back slash). This key is the one to the left of the BACK SPACE key. CTRL- \ tells AmigaDOS that you're finished, and that it should finish writing and close the file. This key represents EOF, for End Of File.

To confirm that you've typed the file correctly, enter

TYPE Hello

You should see the same lines you typed. Now you can start this simple program:

EXECUTE Hello

This should print on the screen:

Hello!
I am your friend, the Amiga
personal computer.

Using ED

It would be nice to have the Amiga actually speak this greeting. Rather than typing in a whole new file, we'll use ED, the screen editor, to make the simple changes we're interested in. Enter

ED Hello

This runs ED and also loads the batch file named Hello. When you start ED, you can give it the name of any file to edit. If the filename doesn't exist, it will be created; otherwise, the file is automatically displayed on the editor screen. (Incidentally, AmigaDOS has another text editor called EDIT, but it's not as easy to use as ED.)

We'll make the Amiga speak the ECHO messages aloud by taking advantage of the system's built-in speech synthesis via the AmigaDOS SAY command (added to AmigaDOS version 1.1). To learn more about SAY, just enter SAY by itself to enter an interactive mode with onscreen instructions.

After you start ED by typing ED Hello, the batch file we previously entered should be on the screen with the cursor at the beginning of the first line. ED is a full-screen text editor, so you can move the cursor anywhere within the file (but not past the last line). To insert some text, just start typing. The DEL and BACK SPACE keys can be used to delete characters.

Move the cursor to the second ECHO line and press RETURN. This inserts a blank line. Cursor up to the blank line and enter

SAY HELLO!

You don't need to press RETURN at the end of the line since you already did this to open up a line for typing.

Now cursor to the end of the file and type

SAY I am your friend, the Amiga personal computer.

(Notice that SAY is the only AmigaDOS command that doesn't require you to enclose text containing spaces with quotation marks.) This is how your screen should look:

```
echo "Hello!"
say Hello!
echo "I am your friend, the Amiga"
echo "personal computer."
say I am your friend, the Amiga personal computer.
```

With the cursor at the end of the file, press the ESC key. An asterisk (*) should appear. Press the X key, then RETURN. This exits ED and saves your changes back to disk.

Finally, type EXECUTE Hello to try out your talking batch file.

Although these techniques are sufficient for simple editing, ED has dozens of editing commands. For example, CTRL-B (press CTRL and B at the same time) blanks out and deletes the line the cursor is on. ESC-J-RETURN joins two lines together. Space doesn't permit a discussion of all these commands, but if you like to experiment, refer to the abbreviated ED reference chart accompanying this article.

Startup-Sequence

A special AmigaDOS batch file, called the *startup-sequence*, is executed automatically when you boot up an AmigaDOS or Workbench disk by inserting it at the Workbench prompt. Startup-sequence normally just displays a message, then launches the Workbench, and ends the command line interface.

To edit this batch file, enter

```
ed s/startup-sequence
```

This runs ED and calls up the file "startup-sequence" from the S subdirectory. This subdirectory, which can also be accessed as the S: device, is a convenient place for batch files. Just as AmigaDOS by default searches for AmigaDOS commands in the C subdirectory, the EXECUTE command first looks for a batch file in the S subdirectory. If AmigaDOS can't find the batch file in this subdirectory, it looks for it in the current directory. So, no matter what your current directory is, you can always use your batch file if you place it in the S directory on your startup disk.

When you first load startup-sequence into ED, you'll see something like this:

```
echo "Workbench disk. Release 1.1"
echo " "
echo "Use Preferences tool to set date"
echo " "
LoadWb
endcli > nil:
```

Since this message appears every time you start up your disk, you may want to change the ECHO statements for a personalized message. Likewise, if you'd rather use AmigaDOS instead of the Workbench, delete the last two lines. The "> nil:" sequence makes AmigaDOS throw away the output of a command; here, the message "CLI task 1 ending."

Startup-sequence is a good place to put personalized commands. For example, if you like to keep your command directory in RAM for speed and convenience, you could insert these lines above the LoadWb line:

```
makedir ram:c
copy c to ram:c all quiet
cd ram:c
```

This copies all of the AmigaDOS commands from the C subdirectory on the floppy disk into a C subdirectory on the RAM disk. It also changes the current directory to the C subdirectory in RAM:, so any AmigaDOS commands you type from then on will be loaded from RAM: instead of from the floppy. In effect, this turns AmigaDOS into a memory-resident DOS, with all commands intrinsic instead of extrinsic. AmigaDOS responds much faster this way. However, this also uses up quite a bit of memory, so you may want to copy only the commands you use frequently.

Another useful startup action is to set the date and time. You can always do this with the Preferences tool or by opening a CLI and using the DATE command. However, it can be more convenient to enter the date when you first turn on your Amiga, allowing all files subsequently saved to be stamped with the current date and time. Just insert this line into startup-sequence:

```
date ?
```

The ? operator can be used in place of the parameter of a command. Instead of specifying the date, ? prompts the user to enter the date. It also displays the template for the date

command (TIME,DATE,TO=VER/K:). If you like, use ECHO to display your own prompt, and > nil: to discard the template:

```
echo "Please enter the date and time."
echo "DD-MMM-YY HH:MM:SS"
date > nil: ?
```

From then on, whenever you boot up from this disk, you'll respond to the prompt by typing something like this:

```
27-jan-86 15:12
```

which automatically sets the system clock.

Variable Parameters

You can also send special options to your batch file. You enter these options on the command line along with the EXECUTE command. Just as with variables in BASIC, you can manipulate these parameters symbolically.

Let's say you'd like a batch file that gives you complete information on a file. It uses LIST to display the information about the file and TYPE to display the file. You would use a command like EXECUTE SHOW RODEO to display the file RODEO. Use ED SHOW or COPY * TO SHOW to create this batch file:

```
.KEY name
LIST <name>
TYPE <name>
```

.KEY (don't forget the leading period) sets up a name for substitution text. Whatever you typed on the same line with EXECUTE is substituted wherever you use <name>. You must use the angular brackets, or LIST and TYPE would look literally for the file "name".

After creating this batch file, type this at an AmigaDOS prompt:

```
EXECUTE SHOW S/STARTUP-SEQUENCE
```

The result is the same as if you had typed LIST S/STARTUP-SEQUENCE followed by TYPE S/STARTUP-SEQUENCE.

Other AmigaDOS commands let you check to see whether the user has entered a specific string and whether a file exists. To prevent an error message, we can check to see if the file exists before we use LIST and TYPE:

```
.KEY name
IF EXISTS <name>
LIST <name>
```

```
TYPE <name>
ELSE
ECHO "<name> does not exist!"
ENDIF
```

Notice the use of IF, ELSE, and ENDIF. Looks like Amiga BASIC, doesn't it? In fact, the AmigaDOS IF-ELSE-ENDIF commands function very much like BASIC's. When the IF condition is true, AmigaDOS executes the following statements; otherwise, the following statements are ignored. ELSE executes the statements following it only if the preceding IF was false. ENDIF cancels conditional processing and returns to executing all commands.

Any Parameters Missing?

Here's how to use the IF EQ option to test for the existence of a command-line parameter. If there is no parameter, <name> is null, so "<name>z" is simply "z". We use NOT to reverse the test. If the parameter "<name>z" is NOT equal to "z", then we must have a command line parameter. (We can't just test IF <name> NOT EQ " ", since EQ wants two parameters, and the null string " " is not a parameter, but the lack of one.)

```
.KEY name
IF <name>z NOT EQ z
LIST <name>
TYPE <name>
ELSE
ECHO "You didn't give me anything to SHOW."
ENDIF
```

Although you can't use leading spaces in the actual batch file, it's easier to follow the IF-ENDIF structures when you use indentation. Just don't type in the leading spaces. This version of the batch file SHOW checks both for the existence of the filename and for the presence of the filename parameter:

```
.KEY name
IF <name>z NOT EQ z
   IF EXISTS <name>
      LIST <name>
      TYPE <name>
   ELSE
      ECHO "<name> does not exist!"
   ENDIF
ELSE
   ECHO "You didn't give me anything to SHOW."
ENDIF
```

You can use more than one parameter in the .KEY statement, just as many commands, such as DATE, accept two inputs.

If the user doesn't enter anything for the parameter, you can assign a default value using either .DEF or $. If you use .DEF, the default phrase is used throughout the batch file. In this example, SHOW displays itself if you don't give it a filename.

```
.KEY name
.DEF s/show
LIST <name>
TYPE <name>
```

You can use $ to substitute a default value only for the current substitution. Several batch commands may use the value in different ways, so each command may have its own default value. In the following example, LIST displays the whole directory if <name> is null, but TYPE types the file "TEMP" if <name> is null:

```
.KEY name
LIST <name>
TYPE <name$temp>
```

Labels and Branching

You can jump forward to a label with the SKIP command. You'd typically use SKIP along with an IF condition if you want to skip over a block of statements that shouldn't be executed if the IF was true. You declare the label with LAB. SKIP can't skip backward, only forward to a LAB statement. You can usually use IF and ELSE to accomplish the same thing, though.

```
.KEY name
IF exists <name>
TYPE <name>
SKIP ToMyLou
ENDIF
ECHO "<name> doesn't exist."
LAB ToMyLou
echo "Finished."
```

An EXECUTE command can execute another batch file, or even itself. This permits backward looping to some degree. Nested batch files can be quite handy. You can test and debug

individual batch programs, then execute them together from a master execute script:

```
EXECUTE Greeting
EXECUTE GetDate
EXECUTE Assignments
```

The individual files could themselves contain other EXECUTE references.

ASSIGNing Shortcuts

If you're using EXECUTE a lot, you may grow weary of typing it. You can always rename EXECUTE to something short like x, but other batch programs may contain EXECUTE statements, requiring you to rename it back. Instead, you can use the ASSIGN command to assign any filename to a device name.

```
ASSIGN x: sys:c/EXECUTE
```

You can now use x: whenever you want to use the EXECUTE command. (The prefix *sys:c/* makes sure that EXECUTE can be found no matter what directory you're in.)

The device name you create should not conflict with an existing one. To get a list of the current assignments, just type ASSIGN. You may want to ASSIGN d: c:list for a convenient and quick shorthand for directories (c: is synonymous with the C directory). You can then just type d: to get a LIST.

ASSIGN can be so handy for this kind of thing that you'll probably want to include your own sequence of ASSIGN commands within startup-sequence. If you put your ASSIGN statement within startup-sequence, you'll get these assignments for every session. Just remember that ASSIGN can be used only to attach a device name to a particular filename. ASSIGN d: "c:list quick" doesn't seem to work. Although LIST is a filename in the c directory, the "quick" parameter is not part of the filename.

Table 4-1. Common ED Commands

Immediate Commands (Hold Down CTRL and Press Key):

CTRL-A	Insert line at cursor position
CTRL-B	Delete current line
CTRL-D	Scroll text downward
CTRL-E	Move cursor to top or bottom of screen
CTRL-N	Delete character at cursor
CTRL-O	Delete word or series of spaces
CTRL-U	Scroll text upward
CTRL-Y	Delete to end of current line

Extended Commands (Precede by Pressing and Releasing ESC):

B	Move cursor to bottom of file
E/*string1*/*string2*/	Exchange *string1* with *string2*
EQ/*string1*/*string2*/	Exchange, but query first
F/*string*/	Find *string*
J	Join current line with next line
Q	Quit without saving text
T	Move cursor to top-of-file
X	Exit, save text

Chapter 5

Graphics

Sheldon Leemon

Graphics

Sheldon Leemon

There are three primary approaches to Amiga graphics. The most fundamental level is via the system hardware, the custom display chips. The next level up is the operating system Kernel graphics routines, which provide the most primitive level of software support for the machine's graphics capabilities. At the top level sits the user interface known as Intuition.

Although Intuition takes care of many facets of the machine's interaction with the user, one of its primary functions is the maintenance of the windowing system that facilitates the multitasking capabilities of the Amiga. While it is quite possible to write programs that directly access the display hardware or use the lowest-level operating system routines, such programs "take over" the machine. They will not be able to coexist with other applications, thus defeating the whole purpose of the Amiga's multitasking operating system (OS).

This chapter will deal primarily with Amiga graphics at the highest level, the graphics which are compatible with Intuition. Most of the examples given will be written in Amiga BASIC, which affords good access to almost all of the OS graphics features supported under Intuition. Since Amiga BASIC provides direct access to the OS Kernel routines, there will also be some discussion of the most useful of those routines as well.

Display Modes

The Amiga graphics hardware offers several ways to adjust the display format. Two levels of horizontal resolution are available, high resolution and low resolution. In high-resolution mode the display is made up of 640 dots of color horizontally. In low-resolution mode the display is only 320 pixels across, each dot being twice as wide as in high-resolution mode.

There is no special "text only" display mode on the Amiga. Text is drawn on the graphics screen. Therefore, the horizontal resolution has a dramatic effect on the size in which this text appears on the screen. In hi-res mode, a maximum of 80 characters can fit into a single line of text when using the system default Topaz 8 font, and 64 characters when

using the Topaz 9 font. In lo-res mode, only half as many characters can fit on a line as in hi-res mode.

There are also two levels of vertical resolution. The standard (noninterlaced) mode provides 200 lines vertically. Interlace mode provides 400 lines of vertical resolution by means of a hardware "trick." Normally, the display is formed by a beam of light that starts at the top of the screen and scans one line at a time from right to left until it gets to the bottom of the screen. Sixty complete scans occur every second. In interlace mode, after every even first scan is finished, the beam of light is moved down half a line so that the odd scans are interlaced between the lines left by the even scans.

Although this interlacing doubles the vertical resolution, it cuts the refresh rate (the number of complete scans per second) in half since it now takes two passes to update the entire display. One result is that in interlace mode the display tends to "flicker," or vibrate. The amount of vibration depends to a large extent on the colors being displayed. An interlaced screen showing black text on a white background is almost unwatchable. The best results are generally obtained when using colors that have as little contrast as possible in brightness levels.

Just as horizontal resolution has an effect on the amount of text that may appear onscreen, so does vertical resolution. Normally, in noninterlaced mode, a maximum of 25 lines of text may appear on the screen when using the system default Topaz 8 font, 22 when using the Topaz 9 font. In interlaced mode, twice as many lines fit onto the screen as in noninterlaced mode.

The third factor to consider in setting up a new display screen is color resolution: the number of colors that can be displayed at any one time. This is determined by the number of bit planes allocated to the display. A bit plane is an area of memory which holds the information concerning which color is to be shown at each dot position of the screen display. Since a bit (or binary digit) can hold either the number 0 or 1, each bit plane increases the number of colors that can be displayed by a factor of two. If one plane is used, each dot can be one of two colors. If two planes are used, up to four colors can appear onscreen at any one time.

The maximum number of colors available depends on the horizontal resolution of the screen. In hi-res mode, up to four

bit planes may be used, for a total of 16 colors onscreen at once. Normally, in lo-res mode up to five bit planes may be used, for a maximum of 32 colors. There are certain special graphics modes in which six planes may be used, but these are esoteric and will not concern most users.

In determining how many bit planes to use, there are a number of tradeoffs to consider. Each bit plane consumes its share of valuable RAM. In lo-res mode, each plane requires 8000 bytes of RAM. High resolution or interlacing doubles the requirement to 16,000 bytes. Using both high resolution and interlacing quadruples the number of bytes used to 32,000 per plane. Therefore, a 640 × 400 display that has four bit planes (allowing up to 16 colors) uses almost *128K* of memory for the display alone. In a 256K system, such a display would consume virtually all of free RAM.

Besides using a lot of memory, hi-res displays that use a lot of bit planes can slow down the microprocessor as well. A good example of this is when you use the built-in speech synthesis while a hi-res screen with four bit planes is being displayed. The voice sounds very scratchy and rough indeed. That's because the job of updating the display places a heavy load on the processor.

Both of these concerns affect the capabilities of the Amiga as a multitasking machine. Obviously, it is going to be much more difficult to run other programs simultaneously if yours leaves no free memory or burdens the processor unduly. This is not to say that you should never use the more memory-consumptive display modes. Rather, you should not use them indiscriminately.

Because graphics displays use so much RAM, it is best to have a 256K RAM expander installed on your Amiga if you plan to program graphics. This is especially true when programming graphics from BASIC. Therefore, some of the more complex BASIC examples in this chapter require more than a 256K system.

Screens

The Amiga's display coprocessor, the copper, allows it to change all of the characteristics of the display on a line-by-line basis. This means that segments of differing horizontal, vertical, and color resolution may appear on the screen at once. These segments of differing displays must be confined to hori-

zontal stripes that extend across the complete width of the screen. While it is technically possible to change display modes in the middle of a horizontal line, such a display is inherently unstable and therefore impractical. In effect, you may not have an area of high-resolution display and an area of low-resolution side by side.

The device that Intuition, the user interface, uses to define screen segments with differing display characterics is known as a screen. A screen must be as wide as the display. Though it can be any number of lines tall, if it is shorter than the display, it must sit at the bottom of the display, not at the top or middle. Since Amiga BASIC does not handle short screens well, we will be dealing only with full-sized screens.

The default screen is known as the Workbench screen. It is a high-resolution, noninterlaced display, which uses two bit planes, for a maximum of four screen colors. This is the display used by the Workbench and CLI (Command Line Interpreter). The color combinations used for this screen are those that have been set with the Preferences program. If none have been set, the Workbench screen uses the system default colors of blue, white, black, and orange.

When you start Amiga BASIC, the BASIC interpreter does not open its own screen. Rather, the Output and List windows are drawn on the Workbench screen. Unless you specify otherwise, all of the output from your program will be displayed in a window having the display characteristics of the Workbench screen.

However, there are several advantages to using the Workbench screen for your programs. It's convenient to use because you don't have to do anything to set it up—it's already there. Also, you don't have to allocate memory for a new screen in addition to the display memory used for the Workbench screen (which will be there in any case). Using the Workbench screen allows easy access to the Workbench or CLI; you just use the depth arrangement buttons in the corner of the window to send your window behind the Workbench. Finally, this arrangement is a reasonably good tradeoff of system resources. It has high resolution for 80-column text and two bit planes for a touch of color, but not enough to hog all of the system's memory.

Still, there will be cases when you'll want to custom-tailor the display characteristics to suit your needs. This means set-

ting up a custom screen. Amiga BASIC provides the SCREEN command for this purpose. The syntax of this command is

SCREEN *screen_number, width, height, depth, mode*

The first value, *screen_number,* is a number from 1 through 4, which is used to identify the screen for the purpose of opening windows (see the section on windows below). The width and height values should be set to the full size of the display. The width should be set to 320 for a lo-res screen and 640 for hi-res. The height should be 200 for noninterlace mode and 400 for interlace mode. Other values will cause strange and unexpected things to happen.

The depth value is a number from 1 through 5 that is used to specify the number of bit planes to use. This in turn determines the number of different colors available at any one time. Each additional plane doubles the number of colors available. The number of colors available for each depth value is as follows:

Depth	Number of Colors
1	2
2	4
3	8
4	16
5	32

Remember that in high-resolution mode, the maximum number of bit planes that can be used is only four. Low-resolution screens can use five bit planes.

The final value to be specified in the SCREEN command is the mode. There are four different display modes available from Amiga BASIC. The meaning of each of the allowable mode values is as follows:

1 Low resolution, noninterlaced
2 High resolution, noninterlaced
3 Low resolution, interlaced
4 High resolution, interlaced

For example, to set up a low-resolution screen without interlacing that can display up to eight colors simultaneously, you could use the command

SCREEN 1,320,200,3,1

When you open a screen with the SCREEN command, BASIC allocates display memory for it. When you are finished

with the screen, you may free up that display memory with the SCREEN CLOSE command, which uses the syntax

SCREEN CLOSE ***screen_number***

In the example above, you could close the screen with the statement

SCREEN CLOSE 1

Windows

A screen can be thought of as the backdrop which defines the conditions for the display. While the screen determines what kind of display is possible, graphics are not normally drawn directly on a screen. Once a screen has been established, it is possible to open up windows on top of that screen and to draw in these windows. Amiga BASIC does not allow you to use a screen until you have opened a window on it.

In Amiga BASIC, graphics output always goes to the window that is currently active. The default is the BASIC Output window, the one that appears on the left side of the screen with the word *BASIC* in its title bar when you start up the BASIC interpreter. But it is possible to open and close your own graphics windows, either on the default Workbench screen or on your own custom screen.

To open a new window, you use the WINDOW command, whose syntax is

WINDOW ***window_num [, [title] [,[size] [,attributes] [,screen_num]]]]***

The first value, *window_num,* is an identification number by which you may refer to that window later on. You use this window number to close the window with the WINDOW CLOSE command and to direct output to it via the WINDOW OUTPUT command.

You may use any number from 1 upward for your window_num. Number 1, however, is reserved for the Output window that BASIC uses. While you can CLOSE this window and reOPEN it like other windows, it still has a special significance since it is the only window in which the user can type immediate-mode BASIC commands. Moreover, a program does not have absolute control over this window, since its comings and goings are affected by the Show Output item on the BASIC menu bar. If there is no Output window currently open, BASIC tends to be fussy about the syntax used to open

one. Since default values do not always apply to the Output window, you may have to specify values that are listed here as optional.

Be careful when using the default Output window for your program's output. Remember that the default window has a size gadget that lets the user change the size of the window at any time. Therefore, if it is important that your window be a certain minimum size (as it almost always is), either open a second window or reopen window number 1 to the requisite size.

A *title* is an optional string expression that will be displayed in the window's title bar. If you omit this expression, there will be no title (and there may not be a title bar, either, depending on the value chosen for attributes). Window 1, the Output window, is an exception. It displays the name of the program, or the word *BASIC* in its title bar if you omit to specify your own title.

Another optional value that you may specify is the size and position of the window. If you omit this value, your new window will cover the entire display screen, except when you open window 1 on the Workbench screen. In that case, the Output window defaults to its previous size.

You specify the size and position of the window by describing the coordinates of the top left and bottom right corners of the window. The format for this description is

(left,top)-(right,bottom)

Recall that the display is 640 pixels (dots) wide in high-resolution mode and 320 pixels wide in low-resolution mode. When describing the left and right coordinates for the screen, horizontal position 0 is at the left edge of the screen, and 319 or 639 is at the right edge, depending on whether low- or high-resolution mode is used. The height of the display is 200 lines noninterlaced or 400 lines interlaced. In describing the top and bottom coordinates, line 0 is at the top of the display, line 199 or line 399 is at the bottom, depending on whether or not the display is interlaced.

This would lead you to believe that the correct description for a full-sized window would be (0,0)–(639,199) for a high-resolution, noninterlaced display. But you must also take into account the space required for the border line that is drawn around the window and for gadgets like the title bar and the

sizing gadget. Because of these requirements, the highest line number that you can specify in your description of the bottom line of the window is 186 for a noninterlaced screen and 386 for an interlaced screen. The highest value that you can specify for the right side of the window is 631 for a high-resolution screen and 311 for a low-resolution screen. If you attach a sizing gadget to your window as described below, this gadget is drawn in the right border of the window and further reduces the possible width of the window. A window that contains a sizing gadget can have a maximum horizontal value of 617 if on a high-resolution screen or 297 if on a low-resolution screen.

To summarize, here are the proper descriptions for the largest possible windows:

(0,0)-(631,186) Hi-res, noninterlaced window without sizing gadget
(0,0)-(617,186) Hi-res, noninterlaced window with sizing gadget
(0,0)-(631,386) Hi-res, interlaced window without sizing gadget
(0,0)-(617,386) Hi-res, interlaced window with sizing gadget
(0,0)-(311,186) Lo-res, noninterlaced window without sizing gadget
(0,0)-(297,186) Lo-res, noninterlaced window with sizing gadget
(0,0)-(311,386) Lo-res, interlaced window without sizing gadget
(0,0)-(297,386) Lo-res, interlaced window with sizing gadget

The attributes value is used to specify which of the standard window gadgets will be attached to the window that you are opening. These include the sizing gadget, the drag bar, the depth arrangement boxes, and the close box. The sizing gadget appears in the lower right corner of the window and allows the user to change the size of the window. The drag bar appears in the title bar at the top of the window; it allows the user to move the window around on the screen if the window is smaller than the screen.

Note that if you don't attach this gadget to the window and don't specify a title for the window, there may be no title bar at all at the top of the window. The depth arrangement boxes appear in the upper right corner of the window, and can be used to send the window to the back of the screen (the dark box) or to bring it to the front of the screen (the light colored box). The close box is located in the upper left corner of the window and allows the user to close the window entirely by clicking on the box.

In addition to specifying which of these gadgets to attach to the new window, the attribute value also lets you choose whether or not you want the contents of the window to be redrawn after the window has been covered by another window or has been resized. This is very convenient, but may be costly in terms of memory usage. If you specify that you wish the window to be refreshed, BASIC must reserve enough memory to save the entire image of the window in memory, regardless of how much of the window is displayed.

Here is a list of the numbers that can be used for the attribute values and their meanings:

Attribute Value	Attribute
1	Window contains a sizing gadget.
2	Window contains a drag bar gadget.
4	Window contains a depth arrangement gadget.
8	Window contains a close gadget.
16	Window contents are saved and redrawn when the window is covered or resized.

Any or all of these values may be used. To attach more than one gadget to your window, add their attribute values together. For example, use an attribute value of 3 if you want a window to have both a sizing gadget (1) and a drag bar (2). Any number 0–31 is a valid attribute value. If you use 0 as the attribute value, you will get a plain window with a border around it (and a title bar, if you have given the window a title). If you do not specify a value here, the default value of 31 (which provides all of the gadgets and screen refresh) is used.

The last option value for the WINDOW command is *screen_num*, the number of the screen upon which you wish the window to be drawn. If you do not specify a value here, the default Workbench screen (whose screen_num is 1) will be used. To place the window on a custom screen, use the screen number that you specified as the first value of the SCREEN command used to create that screen.

When you open a window with the WINDOW command, not only is a new window created, but two other things happen as well. First, this window is brought to the front of the screen and becomes the active window (the window whose title bar is shown in solid lines). Then this window becomes the current Output window. From now on, the output from any

graphics or text commands will be directed to this window, until the program redirects output to another window.

It is possible to make an existing window the current Output window without bringing it to the front of the screen by using the WINDOW OUTPUT command. The syntax of this command is WINDOW OUTPUT *window_num*, where *window_num* is the window number assigned as the first value of the WINDOW command that created the window. It is also possible to bring an existing window to the front of the display and make it the current window with the WINDOW command. The syntax for this form is WINDOW *window_num*, with no other values specified. If the attribute value for that window is 16 or higher, the contents of the window will be restored when the window is brought forward.

When dealing with multiple windows, the WINDOW function can be used to check which window is active, which is the current Output window, the size and depth of the current Output window, and location where the next text character will be drawn. See the dictionary of Amiga BASIC words in Chapter 2 for further details on the WINDOW function.

To close a window, use the WINDOW CLOSE statement. The syntax for it is WINDOW CLOSE *window_num*. When you use this command to close the current Output window, the visible window that was previously the current Output window becomes current once more. Note that this differs from what happens when the user of a program closes a window by clicking on the close gadget. In that case, the closed window remains the current window, and graphics output goes nowhere at all. A program can check to see whether the current Output window has been closed by using the WINDOW function. If WINDOW(7)=0, the current window has been closed, and the program should make another window current.

Color Selection

Now that we have a canvas for our graphics displays, we have only one more subject to discuss before we can start drawing: How do we select the colors we'll want to use?

Remember that the number of colors possible on a given screen depends on the number of bit planes of display memory allocated for that screen. One bit plane allows two colors, two bit planes four colors, three bit planes eight colors, and so

on. In essence, this means that if there are three bit planes, the display memory for each dot position on the display screen can hold a number from 0 through 7. This number does not refer directly to a color, using a code where 0 is black and 1 is white. Instead, the number at the screen dot position refers to a color register.

The color registers may be thought of as a set of 32 pens, each of which may be filled with colored ink in any of the 4096 shades that can be displayed on the Amiga. Register 0 always holds what is normally thought of as the background color; any dot position whose display memory holds the number 0 will display this color. When you wish to use another color to draw a line or a point, you specify the pen (color register) that is to be used in drawing it. Whatever color ink it currently contains is the color that the pen will draw.

Unlike ink, however, the color of a dot drawn on screen can change after you have drawn it. When the display memory for a screen dot holds the number of a particular pen, that dot is whatever color is in the pen at any given moment, not the color that was in the pen at the time the dot was drawn. So, if you use pen 1 to draw a line, and that pen contains the color red, the line will be red. But if you change the color in pen 1 to green after you've drawn the line, the line you drew (and, not incidentally, everything else onscreen that was drawn with pen 1) will instantly become green.

The two factors which determine what color will be drawn on the screen, therefore, are the pen you are using for the drawing and the color used by that pen. You assign a pen number to the Amiga's drawing pens to choose a pen with which to draw. There are two primary drawing pens, the foreground pen and the background pen. The foreground pen is used when drawing single points or solid lines. When you are drawing dotted lines or text, the background pen is used in addition to the foreground pen. Also, the background pen is used to color the entire current Output window when the command CLS is used to clear the screen.

The COLOR command is used to set the color register for each of the drawing pens. The syntax for this command is

COLOR ***[foreground_pen_number] [, background_pen_number]***

where *foreground_pen_number* and *background_pen_number* are the numbers of the pens (color registers) used by the foreground and background pens, respectively. Before a COLOR

statement is issued, the foreground pen defaults to pen 1 and the background pen to pen 0.

The other consideration involves determining the color with which each pen will draw. Colors are chosen by mixing various levels of the colors red, blue, and green. Each color register holds one of 16 color levels for each of these colors. This means that there are 4096 possible colors from which to choose.

In Amiga BASIC, you set the color for each pen with the PALETTE statement:

PALETTE ***pen_number, red_value, green_value, blue_value***

The value *pen_number* specifies the color register whose color you wish to change. The values *red_value, green_value,* and *blue_value* are the levels of each of these three primary colors you wish to use. These are expressed as fractions ranging from 0 (the lowest level, using none of that color) through 1 (the highest level of that color). Although, in theory, the 16 levels could be represented by dividing by 15, in practice, Amiga BASIC does not convert the fractions to color levels quite evenly. The following table shows the values that we will be using with the PALETTE statement and the range of values that can be used to produce the same color level.

Color Level	Palette Value	Range of Acceptable Values
0	0.00	0.00 to 0.03
1	0.05	0.04 to 0.09
2	0.1	0.10 to 0.15
3	0.2	0.16 to 0.21
4	0.25	0.22 to 0.28
5	0.3	0.29 to 0.34
6	0.4	0.35 to 0.40
7	0.45	0.41 to 0.46
8	0.5	0.47 to 0.53
9	0.55	0.54 to 0.59
10	0.6	0.60 to 0.65
11	0.7	0.66 to 0.71
12	0.75	0.72 to 0.78
13	0.8	0.79 to 0.84
14	0.9	0.85 to 0.90
15	1.0	0.91 to 1.00

Since there are 4096 possible combinations, it is impossible to describe each available combination or explain exactly

how to find a particular shade. In general, however, you should remember that the higher the color level, the brighter the color, and the lower the level, the darker the color. Thus, PALETTE 0,0,0,0 sets pen 0 to black, while PALETTE 0,1,1,1 sets it to white. You may lighten a color by raising the two other colors in equal proportions. PALETTE 0,1,0,0 sets pen 0 to a bright red, while PALETTE 0,1,.3,.3 lightens it to a rose color. To darken the original red color, you could try PALETTE 0,.5,0,0.

When you are unsure of what colors to mix, it may help to start with the nearest primary color mixture and experiment. These are the primary color mixtures:

```
PALETTE 0,0,0 'black
PALETTE 0,0,1 'blue
PALETTE 0,1,0 'green
PALETTE 0,1,1 'cyan
PALETTE 1,0,0 'red
PALETTE 1,0,1 'magenta
PALETTE 1,1,0 'yellow
PALETTE 1,1,1 'white
```

If you do not specify a color change for a particular color register, the default color will be used. The default values for each of the 32 pens are listed below:

Pen	Red	Green	Blue	Color
0	0	0.3	0.6	Dark blue
1	1	1	1	White
2	0	0	0.1	Black
3	1	0.5	0	Orange
4	0	0	1	Blue
5	1	0	1	Magenta
6	0	1	1	Cyan
7	1	1	1	White
8	0.4	0.1	0	Dark brown
9	0.9	0.3	0	Red-orange
10	0.55	1	0.05	Lime green
11	0.9	0.7	0	Gold
12	0.3	0.3	1	Blue
13	0.55	0.1	1	Violet
14	0	1	0.5	Blue-green
15	0.75	0.75	0.75	Gray 12
16	0	0	0	Black
17	0.8	0.1	0.1	Red
18	0	0	0	Black

Pen	Red	Green	Blue	Color
19	1	0.75	0.6	Tan
20	0.25	0.25	0.25	Gray 4 (dark)
21	0.3	0.3	0.3	Gray 5
22	0.4	0.4	0.4	Gray 6
23	0.45	0.45	0.45	Gray 7
24	0.5	0.5	0.5	Gray 8
25	0.55	0.55	0.55	Gray 9 (medium)
26	0.6	0.6	0.6	Gray 10
27	0.7	0.7	0.7	Gray 11
28	0.75	0.75	0.75	Gray 12
29	0.8	0.8	0.8	Gray 13
30	0.9	0.9	0.9	Gray 14 (light)
31	1	1	1	White

Keep in mind that the same color palettes are used by every window in a screen. When you change the pen colors with the PALETTE statement, you affect the color of every window that appears in the same screen as the current Output window. The change is limited to that screen, however, and windows in other screens will not be affected.

Drawing Points

Now we're ready to draw something. The simplest drawing commands, PSET and PRESET, color a single dot on the screen. The two commands are identical except that PSET uses the foreground pen by default, and PRESET uses the background pen. The syntax for these commands is

PSET [STEP] *(x,y) [,pen]*
PRESET [STEP] *(x,y) [,pen]*

There are two ways to indicate where you want the dot drawn. The first is to use absolute horizontal and vertical coordinates. The horizontal coordinates range from 0 at the left edge of the screen through a maximum of 631 or 311 at the right edge of the screen, depending on whether your screen is high resolution or low resolution. Remember that if you have a sizing gadget in the right border of the window, the maximum is cut to 617 or 297. The vertical coordinates range from 0 at the top of the screen through 186 or 386 at the bottom, depending on whether the screen is noninterlaced or interlaced. If there is no title bar, the coordinate at the bottom of the screen is 195 (noninterlaced) or 395 (interlaced). So, to put a white dot (the default color of the default foreground pen)

midway down the left edge of the standard Output window on the Workbench screen, you would use the command

PSET (0,98)

To erase that dot (by drawing over it with the background pen), you could use the command

PRESET (0,98)

The other way to specify the point at which to draw the dot is to indicate that you wish to use relative coordinates by including the keyword STEP. Relative coordinates specify a position relative to the last dot drawn. If none have yet been drawn, the position is relative to the middle of the Output window (including the borders). For a full-sized, low-resolution, noninterlaced window, for example, this position would be (160,100). A positive horizontal coordinate indicates that the dot will be to the right of the last one, while a negative coordinate positions the dot to the left. A positive vertical coordinate means that this dot will be lower than the last one, and a negative vertical coordinate means it will be closer to the top. For example, if the last dot drawn was at (100,50), the command

PSET STEP (−10,20)

would draw the next dot at (90,70). If the last dot drawn was at position (150,90), the command

PSET STEP (−40,−10)

would draw a dot at (110,80). Relative coordinates are extremely useful when you want to draw the same image in a number of different places, or if you aren't quite sure where the image will be drawn.

Let's say, for example, that you are drawing an image in a window that has a sizing gadget. If the user leaves the window alone, the right edge may be at position 600. But if he or she shrinks the window, its right edge may be only at position 400. You can find the right edge with the WINDOW function and set the first point accordingly. By using relative coordinates for the rest of the drawing statements, they will all then be positioned properly, regardless of where the right edge of the window is.

The other advantage of using relative coordinates is that they can make it easier to change your program. If you later decide that you want to move an image a few pixels over from

its original location, it is much easier just to change the starting point than it is to change the coordinate for every point.

Both the PSET and PRESET commands take an optional pen value. That value, if specified, selects the pen to be used in drawing the dot. If none is specified, the PSET command uses the color register associated with the foreground pen, and PRESET uses the color register associated with the background pen. The foreground and background pen values default to color registers 1 and 0, respectively. These assignments can be changed at any time, however, by using the COLOR statement (see above). Note that when you specify which pen to use, PSET and PRESET can be used interchangeably since the only difference between them is the default pen that each uses.

Sometimes it is useful for a program to be able to tell which pen was used to color a particular location in a window. Amiga BASIC provides the function POINT (x,y) which is the same as the operating system routine called ReadPixel. For the purposes of demonstrating how to use an operating system graphics routine, we'll use ReadPixel.

From BASIC, you may access an operating system graphics routine by using the LIBRARY statement to let BASIC know how to find the routine and how to interact with it. In this case, the correct form of the statement is

LIBRARY "graphics.library"

When this LIBRARY statement is used, BASIC gets information about the location of the system graphics routines from a file called "Graphics.bmap". This file is included on the Amiga BASIC disk, in the BasicDemos directory, and must be present in the current disk directory when the program containing the LIBRARY statement is run.

Since ReadPixel is a function that returns the number of the pen used to color a particular dot, you must also use the DECLARE FUNCTION statement to let BASIC know that it must search the graphics library for this function. To declare the ReadPixel function, use the command

DECLARE FUNCTION ReadPixel&( ) LIBRARY

Note that we refer to the routine as ReadPixel&. The ampersand following the name of the routine indicates that ReadPixel is a long integer. This specification is necessary because BASIC places the address of the operating system routine in the variable ReadPixel&, and an address cannot be

longer than a long integer. If you forget to make the name of the routine a long integer, BASIC will return a "Type Mismatch" error.

Now we are ready to call the ReadPixel routine. The proper syntax by which to call this routine is

Pen& = ReadPixel& (RastPort&,x&,y&)

The value *RastPort&* is the address of the window's RastPort, a data structure that the operating system uses to keep track of graphics in that window. The address of that structure is returned by the function WINDOW(8). The x& and y& values stand for the horizontal and vertical coordinates of the point that we want to read. Just as it is important to indicate the variable type in the name of the routine, it is also important that the values passed to the routine are of the correct type. In general, these values should be integers.

The following short sample program puts all of the steps together and shows how to read the pen value used at a particular coordinate:

```
LIBRARY "graphics.library"
DECLARE FUNCTION ReadPixel&( ) LIBRARY
RP&=WINDOW(8)          'Find RastPort address
PSET (100,50),3        'Draw at 100,50 with Pen 3
Pen&=ReadPixel&(RP&,100,50) 'Read dot at 100,50 into Pen&
PRINT Pen&             'Should be 3, for Pen 3
```

This program draws a dot at (100,50) with pen 3, then reads the pen value at (100,50) into the variable Pen&. The value of Pen& is then printed to confirm that the pen number has been correctly read.

Drawing Shapes

Drawing single points is the least of the Amiga's abilities. Amiga BASIC and the operating system also have commands that allow you to draw lines and entire geometric shapes such as rectangles, squares, circles, ellipses, and polygons.

You use the LINE command to draw lines or rectangles. The syntax of this command is

LINE [[STEP] *(x1,y1)***] - [STEP]** *(x2,y2), [pen_number]* **[,b [f]]**

You must specify two pairs of coordinates for the LINE command, one for the starting point and one for the ending point. These coordinates can be absolute coordinates, relative co-

ordinates, or a combination of absolute and relative. For example, the statements

```
LINE (30,50) - STEP (40,40)
LINE STEP (0,0) - STEP (-40,40)
```

first draw a line from (30,50) to (70,90), and then draw a line from that point to (30,130).

The value *pen_number* can be used to indicate the pen to use in drawing the line. If no pen is specified, the foreground pen is used as the primary drawing pen.

Besides drawing lines, the LINE command can be used to draw rectangles. By adding the letter *b* after the pen number, you can indicate that you want a box to be drawn. When this option is invoked, the first pair of coordinates indicates the top left corner of the box, while the second pair determines where the lower right corner will be placed. For example, the statement

```
LINE (100,50) - STEP (50,50),,b
```

draws a box from (100,50) to (150,50) to (150,100) to (100,100) to (100,50), using the foreground pen color.

If you use the letters *bf* instead of just *b*, the box will be filled in with the color selected. We'll discuss filled shapes below.

To review what we've presented so far, here are a couple of short example programs. The first uses the LINE and PSET commands to draw three lines in different colors. It uses the default Output window that sits on the 640 × 200 Workbench screen.

```
LINE (50,50)-STEP (100,50)          'draw first with foreground pen (1)
LINE STEP (0,0)-STEP(-100,50),3     'second line drawn with pen 3
FOR y = 50 TO 150
  PSET (50,y),2                     'third line with pen 2
NEXT
END
```

The second program draws the same three lines, but this time in a new window on a low-resolution screen which has its own custom color palette. It also uses the ReadPixel routine to read each dot in a rectangular area that contains parts of the three lines and then resets each point to a new color. When the drawing is done, the program waits for the user to click the mouse button and then closes the new window and screen.

```
LIBRARY "graphics.library"
DECLARE FUNCTION ReadPixel&( ) LIBRARY

SCREEN 1,320,200,2,1  '320 × 200 lo-res, 4-color screen
WINDOW 2,,,0,1        'Full-screen window, no gadgets
RP& = WINDOW(8)  'This window's RastPort address

PALETTE 0,1,1,1  'White background
PALETTE 1,1,0,0  'red
PALETTE 2,0,1,0  'green
PALETTE 3,0,0,1  'blue

LINE (50,50) - STEP (100,50)          'draw first with foreground pen (1)
LINE STEP (0,0) - STEP (-100,50),3  'second line drawn with pen 3
FOR y = 50 TO 150
  PSET (50,y),2                       'third line with pen 2
NEXT

FOR y& = 50 TO 150
  FOR x& = 49 TO 100
    Pen& = ReadPixel&(RP&,x&,y&) 'read pen for each point
    PSET (x&,y&), 3-Pen&         'and complement
  NEXT x&
NEXT y&

WaitForClick: IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2   'close the window
SCREEN CLOSE 1   'and the screen
END
```

One of the graphics library routines, PolyDraw, can be helpful in drawing shapes by connecting a number of lines. To use this routine from BASIC, you must first open the graphics library with the statement

LIBRARY "graphics.library"

as explained above. Once the library is open, you may call PolyDraw with the BASIC statement

CALL PolyDraw& (RP&, *coordinate_pairs, array_address)*

where RP& is the RastPort address of the window, WINDOW(8). The other values specify the points to be drawn. To use PolyDraw, you must first set up an array of short integers. This array must hold the coordinates of each point which is to be connected by a line. For example, if you wanted to use PolyLine to draw a line from the current pixel position to (100,100), then to (120,70), then to (90,50), you could set up an array POINTS%( ) where POINTS%(0)=100, POINTS%(1)=100, POINTS%(2)=120, POINTS%(3)=70, POINTS%(4)=90, and

POINTS%(5)=50. You would then call PolyDraw with the statement

CALL PolyDraw& (RP&,3,VARPTR(POINTS%(0))

The number 3 indicates that there are three pairs of coordinates. It is important to remember that the proper value for the coordinate_pairs value is not the size of the array, but the number of coordinate pairs (half the size of the array). The second value to pass is the address of the array, which can be found by using the VARPTR function.

PolyDraw uses the current location of the pixel cursor as its starting point. This location depends on where the last point was drawn; if none were drawn, it defaults to (0,0). Rather than leaving things to chance, you will probably want to move the pixel cursor to the correct starting location before calling PolyDraw. This can be accomplished with a call to Move, another operating system routine. Move merely sets the pixel cursor without doing any drawing. Here's the proper way to call this routine from BASIC (the graphics library must be opened first):

CALL Move& (RP&, x&,y&)

RP& is the address of the window RastPort—WINDOW(8)—and x& and y& are the horizontal and vertical coordinates at which the pixel cursor is set. Although Move appears to have no effect on relative coordinates used with LINE and PSET (BASIC appears to keep its own internal pixel cursor), it has a definite effect on the positioning of BASIC text. Preceding a PRINT statement with a call to Move allows you to position text at precise coordinates rather than at a particular character position.

The following short program shows how to draw an eight-sided figure using PolyDraw. It uses a custom low-resolution screen and shows how to use the COLOR command with CLS to clear the window to a particular color.

```
LIBRARY "graphics.library"

SCREEN 1,320,200,4,1    '320 × 200 lo-res, 16-color screen
WINDOW 2,,,0,1          'Full-screen window, no gadgets
Rp&=WINDOW(8)           'Window's RastPort address
COLOR 9,1               'foreground to red, back to white
CLS                     'clear screen to white
```

```
DIM points%(16)
FOR p=0 TO 15
  READ d
  points%(p)=d          'put coordinate pairs in array
NEXT
  DATA 180,50, 210,80, 10,120, 180,150
  DATA 100,150, 70,120, 70,80, 100,50
CALL Move& (Rp&,100,50)                          'move pixel cursor
CALL PolyDraw& (Rp&,8,VARPTR(points%(0))) 'draw polygon

WaitForClick: IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

The BASIC command used to draw circles, ellipses, and arcs is the CIRCLE command. The syntax of this command is

CIRCLE [STEP] ***(x,y),radius [,pen_number [,start_angle,end_angle [,aspect_ratio]]]***

The only values that are required are the coordinates of the center point and the radius of the circle. The center coordinates may be expressed as an absolute, for example, (50,50), or relative to the point where the last dot was drawn, for example, STEP (50,50)—50 dots to the right and 50 dots below the last position drawn. After the circle is drawn, the pixel cursor remains at the center point of the circle, even though no dot is drawn there. This means that you can draw concentric circles by specifying a center point of STEP (0,0) for each circle, as the following program demonstrates:

```
CIRCLE (100,100), 20
FOR R=27 TO 100 STEP 7
  CIRCLE STEP (0,0), R
NEXT
```

The radius value is the radius of the circle, expressed in pixels. Note that the vertical radius will not be the same as the horizontal radius. Since there are 640 pixels across and only 200 lines vertically on the Workbench screen, a circle that was 100 pixels wide and 100 pixels high would be tall and skinny, not round. Therefore, the CIRCLE command automatically scales down the vertical radius to make the circle appear round. You can change this scaling with the aspect_ratio value, discussed below.

The pen_number value designates the pen that you want

used to draw the circle. If you do not specify a pen number, the foreground pen is used as a default.

The start_angle and end_angle values allow you to draw only a portion of the circle or ellipse. The values designate the starting and ending angles of the arc, expressed in radians. Since there are 2π radians in a circle, the permissible values range from 0 through 2π. The point described by a value of 0 is the rightmost point on the circle. As the value increases, you move around the circle counterclockwise. The value for the point at the top of the circle is $\pi/2$, that for the left of the circle is π, and that for the bottom of the circle is $3\pi/2$. To convert degrees to radians, use the formula

radians = degrees/180 * π

The starting angle may be smaller than the ending angle, but in either case, the arc will be drawn counterclockwise. This means that the statement

```
CIRCLE (100,100), 70,, 0, 3.14*3/2
```

draws three-quarters of a circle (starting at the right and moving counterclockwise to the bottom). If you reverse the starting and ending points, however, the statement

```
CIRCLE (100,100), 70,, 3.14*3/2, 0
```

draws only a quarter circle (starting at the bottom and moving counterclockwise to the right side).

If either the start_angle or end_angle value is negative, the value will be treated as if it was positive, but that position on the arc will be connected by a line to the center of the circle. For example, the statement

```
CIRCLE (100,100), 70,, −3.14*7/4, −.01
```

produces a pie-shaped wedge with both ends of the arc connected to the center point. Reversing the starting and ending points gives you a circle with a wedge cut out of it.

The aspect_ratio value describes the scaling used to make the circle appear round or elliptical. Since the standard Workbench screen is 640 pixels wide, but only 200 lines high, a circle which is as many dots tall as it is wide will be tall and skinny, instead of round. So the width is multiplied by the aspect_ratio value to determine the height. The default value for a high-resolution, noninterlaced screen is 0.44, which means that the vertical radius will be 44 percent as large as the horizontal radius. For a low-resolution, noninterlaced

screen, the default is 0.88, twice as large, because the screen is half as many dots wide. An aspect_ratio value of less than the default creates an ellipse that is short and fat, while a value that is greater than the default creates an ellipse that is tall and skinny. If the aspect_ratio value is greater than 1, the horizontal radius may be shortened to preserve the ratio of height to width. This means that the ellipse created by the statement

CIRCLE (100,100), 70,,,, 10

will be narrower than the circle drawn by the statement

CIRCLE (100,100), 70

Patterned Lines

So far, all our lines have been solid. The Amiga graphics hardware, however, is capable of drawing dotted lines as well. The pattern for line drawing is set with the PATTERN command:

PATTERN ***[line_pattern] [,area_pattern]***

The value which determines how lines are drawn is *line_pattern.* The other value, *area_pattern,* is used for pattern fills and will be discussed below in the section on filled shapes.

The line_pattern value is an integer expression that describes a mask that is 16 dots wide. For example, the number 65,535 in base two (binary numbers) looks like this:

1111111111111111

If you imagine this as a pattern for line drawing, where every one represents the position of a dot drawn with the foreground pen and every zero represents the position of a dot drawn with the background pen, you can see that this pattern will produce a solid line drawn with the foreground pen. The number 43,690, on the other hand, looks like this:

1010101010101010

This pattern alternates one dot of foregound color with one dot of background color.

Counting in binary does not come as second nature to most of us. Hexadecimal is somewhat easier to use when figuring out line patterns, since each digit corresponds to four dots. The following table shows the correspondence between dot patterns and hexadecimal digits:

Hex Digit	Dot Pattern
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

By breaking the 16-dot-wide mask down into 4-dot groups, we can figure out the patterns a little more easily. For example, if we want a pattern where 3 dots of foreground color alternate with 1 of background, we choose the pattern that corresponds to E hexadecimal and repeat it four times. The proper BASIC statement for such a pattern would therefore be

```
PATTERN &HEEEE
```

Patterns apply to any BASIC or operating system commands that use the hardware line-drawing capabilities. This includes the LINE statement and PolyLine graphics routine. The line pattern also affects the lines that are used to connect the starting and ending arcs to the center when negative values are used for *start_angle* and/or *end_angle* in the CIRCLE command. The following short program shows the effect of PATTERN on these commands:

```
LIBRARY "graphics.library"

WINDOW 1,,(0,0)-(617,186),31,-1 'Make Output window full size
Rp&=WINDOW(8)                   'Window's RastPort address

DIM points%(16)
FOR p=0 TO 15
 READ d
 points%(p)=d
NEXT
 DATA 180,50, 210,80, 210,120, 180,150
```

```
DATA 100,150, 70,120, 70,80, 100,50

PATTERN &HFF00                              'even stripes
COLOR ,2                                    'white and black
CALL Move& (Rp&,100,50)
CALL PolyDraw& (Rp&,8,VARPTR(points%(0)))   'draw polygon

PATTERN &HFFF0                              'mostly foreground
LINE (250,10) - STEP (350,170),3,b          'draw orange and black
                                            box

PATTERN &HAAAA                              'dotted pattern
COLOR ,3                                    'white and orange
LINE (70,10)-(210,40)                       'draw dotted line

PATTERN &HF0F0                              'smaller stripes
COLOR ,0                                    'default colors
CIRCLE (425,95), 100,1, -4.71,-3.14         'circle with wedge
                                            removed

END
```

When you want to switch back to drawing a solid line, use the command PATTERN −1.

Drawing Modes

Until now we have been doing all of our drawing in BASIC's default drawing mode. This mode is known as the JAM2 mode, because it "jams" the color of the foreground pen and the color of the background pen into display memory at the same time. Since the default color for the background pen is the same as that of the display background, it is sometimes difficult to tell that two colors are being drawn at once. But if you enter the command

```
COLOR ,3
```

in immediate mode, it is quite easy to see that text is printed in two colors (in this case, white on orange). JAM2 mode is also evident in patterned line drawing. In JAM2 mode, all of the bits of the line pattern that are set to 1 are drawn with the foreground pen, and all of the bits of the line pattern that are set to 0 are drawn with the background pen. When drawing solid lines, the JAM2 mode does not quite live up to its name, since only the foreground pen is used.

There is another major drawing mode on the Amiga that may be accessed from BASIC by using the operating system graphics library. This mode is known as JAM1 mode. As you might have guessed, in JAM1 mode only one pen, the fore-

ground pen, is used for drawing. This means that the area that would normally be drawn with the background pen in JAM2 mode is left undisturbed.

With JAM1 mode, it is possible to superimpose text on a graphics image without blotting out a rectangle of that image. Patterned lines will turn out differently in JAM1 mode from those in JAM2 mode, since only the bits of the pattern that contain ones will be colored. Solid lines are drawn the same as in JAM2 mode, however, since only the foreground pen is used in either case.

There are also two modes which modify the drawing modes. The first is known as COMPLEMENT mode. In this mode, neither the foreground nor background pen is used. Instead, the color of each screen dot where a pen was supposed to draw is complemented. Complementing the color of a pixel inverts the bits of its pen number, changing all the ones to zeros and all the zeros to ones.

For those who do not think in binary, another way of looking at the process is that you take the highest possible pen number, subtract the pen number of the current color, and you are then left with the pen number of the new color. For example, if the window that you're drawing on is attached to a screen that has three bit planes, there are eight drawing pens. These pens are numbered 0–7. If you want to find the complement of pen 2, subtract 2 from 7, which leaves 5. If you are using four bit planes, the highest pen number is 15, so the complement of pen 2 would be pen 13.

COMPLEMENT mode is really useful only with JAM1 mode. If you add COMPLEMENT mode to JAM2 mode, areas that would be drawn with the background pen are complemented along with areas that are drawn with the background pen. The result is that the entire area is complemented, and you end up with solid lines instead of patterned lines, and thick solid lines instead of text.

The other mode that can be used to modify JAM1 and JAM2 is called INVERSVID. Used mostly for text, INVERSVID reverses the roles of the foreground pen and background pen. Text that is drawn in JAM1 mode with INVERSVID has the background area surrounding the letters colored in with the foreground pen, while the area of the letters themselves is left untouched. If COMPLEMENT mode is added to JAM1, the areas that normally would be colored with the background

pen are the ones that are complemented. Used with JAM2 mode, INVERSVID merely reverses the colors of foreground and background pens.

These drawing modes can be set with the operating system routine SetDrMd (Set Draw Mode). The sytnax for SetDrMd is

CALL SetDrMd& (RP&,Mode&)

where RP& is the address of the window's RastPort—WINDOW(8)—and Mode& is the value equal to the modes desired. Use the following table to find the correct value for the modes desired.

Mode	Value
JAM1	0
JAM2	1
COMPLEMENT	2
INVERSVID	4

You may add these values to form any combination. To select a combination of JAM1, COMPLEMENT, and INVERSVID modes, for example, you would use the number 6 (0+2+4).

The following program should help you visualize the effects of the various drawing modes:

```
LIBRARY "graphics.library"
DEFLNG A-Z          'all long integers
SCREEN 1,320,200,3,1 '320 × 200 lo-res, 8-color screen
WINDOW 2,,,0,1       'Full-screen window, no gadgets

PALETTE 0,1,1,1  'White background
PALETTE 1,1,0,0  'red—used for foreground pen
PALETTE 2,0,1,0  'green—used for background pen
PALETTE 3,0,0,1  'blue
PALETTE 4,1,1,0  'yellow—complement of blue
PALETTE 7,0,0,0  'black—complement of white

Mode$(0)="JAM1 mode"
Mode$(1)="INVERSVID JAM1"
Mode$(2)="JAM2 mode"
Mode$(3)="INVERSVID JAM2"
Mode$(4)="COMPLEMENT mode"
Mode$(5)="INVERSVID COMP"

LINE (92,17) - STEP (170,120),3,bf    'draw blue box for contrast
PATTERN &HFF00                        'striped pattern
COLOR 1,2                             'set colors to red and green
```

```
FOR Row = 0 TO 5                          'For six lines
  y=((Row \2)+1)*5+(Row MOD 2 = 0)
  LOCATE y,4                              'Position for PRINT
  Mode = (Row \2) - 4*(Row MOD 2 = 1)
  CALL SetDrMd& (WINDOW(8),Mode)          'Set drawing mode
  PRINT "This is ";Mode$(Row)             'Print text in this mode
  LINE (219,y*8-4)- STEP (85,0)           'Draw line in this mode
NEXT

WaitForClick: IF NOT MOUSE(0) THEN WaitForClick
WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

Filled Shapes

Not only can the Amiga graphics routines draw lines and shapes, but they can fill them with color as well. We have already seen one example of filled shapes in the LINE command. As you may remember, if you add the letters *bf* to the end of this command, a filled box is drawn. For example, this line draws a box that is 200 pixels by 100 pixels, using the color of the foreground pen:

```
LINE (20,10) - STEP (200,100),,bf
```

Other Amiga BASIC commands assist you in drawing filled shapes also. The AREA and AREAFILL commands are used to draw filled polygons, much like the graphics routine PolyDraw is used to draw open polygons. PolyDraw requires you to put the coordinate pairs that define the shape in an array, but AREAFILL requires that you specify each point individually with the AREA command. The syntax of this command is

AREA [STEP] *(x,y)*

The only value that you must specify is a coordinate for one of the points of the filled polygon. This coordinate may be expressed as an absolute position, such as (10,20), or relative to the last point drawn, STEP (10,−20).

To draw a filled polygon with AREAFILL, issue an AREA command for each point of the polygon in the order in which you want the points drawn. You do not have to specify the starting point twice since the last point will automatically be connected to the first point. A maximum of 20 points may be used to define the polygon. If more AREA statements are used, all but the first 20 are ignored. When enough AREA

commands have been given to cover all of the points in the polygon, use the AREA command to connect the points and fill the polygon.

The following program shows how to draw a filled version of the eight-sided figure used in the PolyDraw example.

```
FOR p=0 TO 7
 READ x,y
 DATA 180,50,  210,80, 210,120, 180,150
 DATA 100,150, 70,120, 70,80,   100,50
 AREA (x,y)   'AREA for each coordinate pair
NEXT
```

AREAFILL 'draw filled shape

The last of the shape filling commands is a general-purpose floodfill command called PAINT. Unlike the two previous commands that we've discussed, PAINT does not draw the shape and then fill it in. PAINT colors in an existing enclosed area. The syntax for the PAINT command is

PAINT [STEP] ***(x,y) [,fill_pen [,border_pen]]***

The only required value is the coordinates of the point at which the filling begins. This coordinate pair may be expressed as an absolute location or relative to the location of the last dot that was drawn.

The two optional values that you may specify are *fill_pen*, the number of the pen which is used to do the filling, and *border_pen*, the number of the pen at which the filling stops. The default value for the fill_pen is that of the foreground pen, while the border_pen defaults to the same value as is currently in fill_pen.

PAINT is used to fill a shape that is entirely enclosed by a border color. It starts drawing at the specified coordinates and spreads out in all directions, filling every dot that is not drawn in the border color. As a consequence, if the shape that you choose to PAINT is not completely enclosed by the border color, the fill color will escape through the gap and spread out to cover the entire window. Likewise, if you have not specified a border_pen that matches the border color, the fill will proceed right through the border.

There are some more serious concerns associated with using the PAINT command. The command will not work with a window set for "smart" refreshing of the screen. If you try

to use PAINT in a window which was opened with an "attribute" value greater than 15, you will crash the system. Since the default Output window has an attribute value of 31, it is not safe to use PAINT in that window, unless you reopen it with a WINDOW command giving a lower attribute value. Another point to watch for is specifying coordinates for PAINT that lie outside the window boundaries. This is particularly easy to do when you are specifying relative coordinates. Such a PAINT command "fills" areas of memory that do not belong to the display, and this can also crash the system.

The following example draws a circle, PAINTs it white, and PAINTs the rest of the window orange.

```
WINDOW 1,,(0,0)-(300,186),15  'Reopen Output window to type 15
CIRCLE (150,100),100          'draw the circle
PAINT STEP (0,0)              'fill it with foreground pen
PAINT (0,0),3,1               'fill rest of screen with pen 3
```

Patterned Fills

Just as the PATTERN command can be used to set a pattern for line drawing, it can also be used to set a pattern for area filling. The process of setting up the fill pattern is a little more complex, since a two-dimensional area is involved. The pattern is still 16 bits wide, but it is several lines high as well. You can choose the height of the pattern yourself, as long as you stick to a power of 2 (2 lines, 4 lines, 8 lines, 16 lines, and so on). Since you are working with a screen with a maximum height of 200 lines, don't make the pattern more than 64 lines high.

The area fill pattern should be stored in an array of 16 bits (short integers). First, DIMension the array to a power of 2. For example, the proper DIM statement for an eight-element array called Pattern% is

```
DIM Pattern%(7)
```

since the array starts with element 0.

Next, you must determine the values with which to fill the array. You go about this in the same way as in determining the line pattern values. It may help to visualize the pattern if you write it out in binary digits, using ones to stand for dots filled with the foreground color and zeros to stand for dots filled with the background color. For example, let's look at a pattern that draws the letters *HI*.

```
0000000000000000 = &H0000
0110011001111110 = &H667E
0110011000011000 = &H6618
0111111000011000 = &H71E8
0110011000011000 = &H6618
0110011000011000 = &H6618
0110011001111110 = &H667E
0000000000000000 = &H0000
```

Once we have determined the values for the pattern elements, we assign these values to the array Pattern%. Then, we set the area fill pattern to the values stored in this array with the statement

PATTERN ,Pattern%( )

When the pattern is set, BASIC makes its own internal copy of the array. Pattern% is no longer needed, unless you want to set the pattern to another array and change it back later. You may ERASE the array after the PATTERN command is given in order to free up memory.

This sample program fills a box with the "HI" pattern that we designed above:

```
WINDOW 1,,(0,0)-(250,186)       'size the Output window

DIM pat%(7)
FOR p=1 TO 6
  READ d
  pat%(p)=d                     'put the pattern in an array
NEXT
  DATA &h667e, &h6618, &h7e18
  DATA &h6618, &h6618, &h667e

PATTERN ,pat%                   'use the pattern for fills
LINE (16,32)-STEP(192,96),,bf   'draw a filled box

FOR p=0 TO 7
  pat%(p)=-1
NEXT
PATTERN -1,pat%                 'return the pattern to solid
END
```

Notice that at the end of the program, we set the area fill pattern back to a solid pattern. If we had not done so, the cursor in our default Output window would have been rendered difficult to see. We do not have to change the pattern back at the end of the next example because it opens its own

window rather than using the default window. Each window has its own private line pattern and area fill pattern.

The area fill pattern is used with all of the commands that produce filled shapes. The following program demonstrates the three different kinds of patterned fills: boxes, AREAFILLs, and PAINTing.

```
SCREEN 1,320,200,4,1          '16-color, lo-res
WINDOW 2,,,0,1                'full-sized window

PALETTE 0,0,0,0               'black background

DIM pat%(7)                   'pattern array has 8 elements

COLOR 9,14
f=0:GOSUB Fillpat             'set pattern
CIRCLE (160,100),70
PAINT (160,100)               'flood fill

COLOR 5,6
f=&HA5A5:GOSUB Fillpat        'set pattern
LINE (10,10)-STEP(60,90),,bf  'rectfill

COLOR 11,9
f=&H5555:GOSUB Fillpat        'set pattern
 AREA (280,100)
 AREA STEP (0,80)
 AREA STEP (-80,0)
 AREAFILL                     'area fill

WaitForClick: IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
WINDOW OUTPUT 1
END

Fillpat:      'create a repeating or random fill pattern
RANDOMIZE TIMER
FOR p=0 TO 7
  IF f=0 THEN pat%(p)=RND*60000&-30000 ELSE pat%(p)=f
NEXT
PATTERN ,pat%
RETURN
```

Text

As we have seen before, text is drawn on the graphics screen like any other image. This means that graphics and text images may be mixed freely. The various drawing modes (see

above) affect text output. And there are some other ramifications of text being treated like any other graphics image as well. For one thing, the programmer has some responsibility for making sure that the text is printed within the confines of the window in which he or she is working. Try typing the following one-line BASIC program in the immediate mode:

```
FOR x=0 TO 255:?x;:NEXT
```

You will see that the PRINTed output goes right off the edge of the screen. To insure that this does not happen in your program, you may use the WIDTH command to set the maximum line width. When used for this purpose, the syntax of the command is

WIDTH *[linesize] [,print_tab]*

where *linesize* is the maximum line length, and *print_tab* is an optional value that specifies the width of the columns used when a comma is added to the end of a PRINT statement. The maximum line length depends both on the width of your window and the size of the text font that you are using. If you have set the 80-column font as the default using the Preferences program, then each character will be eight dots wide, resulting in a maximum line width of 80 characters for a hi-res window and 40 for a lo-res window. Actually, since some room is taken up by the border drawn around the window, and the sizing gadget (if present), this will be reduced to 75 or 76 characters in hi-res and half that number in lo-res. If you have set the 60-column font as the default, then the width of each character will be ten pixels, and the maximum number of characters must be reduced accordingly.

A related problem is how to tell what size of text will be used. The default character size is determined by the settings of the Preferences program, so there is no clear way to tell whether BASIC will start up in the 60-column (9 point) or 80-column (8 point) font. Since each letter of the former font is larger than the corresponding character of the latter, text that has been carefully positioned for one mode might appear completely out of line in the other.

One way around this problem is to have your program open a new window and specify the font to be used in that window. To specify the font to be used in a window, you must use the operating system routines OpenFont, SetFont, and CloseFont. The first step is use OpenFont to get a pointer

to a font descriptor. Since this call returns a value, you must use the DECLARE FUNCTION statement as well as opening the graphics library with the LIBRARY statement. The proper syntax for this call is

```
FontPtr& = OpenFont&(VARPTR(textAttr&(0)))
```

The one value that must be supplied to the OpenFont command is the address of a long integer array that holds a description of certain text attributes. The first element of the array holds the address of a text string, ending with an ASCII 0, that names the font. In the case of the system fonts, the name is "topaz.font". The other element of the array holds the height of the font and some additional information about the font, which for the system font equals zero. The form of the text attribute array therefore is

```
textAttr&(0) = SADD( "topaz.font"+CHR$(0) )
textAttr&(1) = height*65536&
```

where *height* is either 8, for the 80-column font, or 9, for the 60-column font.

Once FontPtr& for a particular font has been found, it can be used to set that font for use in a particular window with the SetFont call. The syntax of that call is

```
CALL SetFont&(RP, FontPtr&)
```

where RP is the address of the window's RastPort structure—found by the function WINDOW(8)—and FontPtr& is the pointer found by the OpenFont call.

Finally, when you are through with a particular font, you should close it with the CloseFont call:

```
CALL CloseFont&(FontPtr&)
```

Since text is "drawn" on the graphics screen, it is quite possible to mix fonts in one window. The following program opens a window and writes one sentence in each of the two system fonts:

```
'This program prints both system fonts

DEFLNG a-z      'all variables default to long integer
DECLARE FUNCTION OpenFont LIBRARY

LIBRARY "graphics.library"
WINDOW 2,"System Fonts",(100,50)-(525,100),12
WIDTH 41
```

```
FOR height=8 TO 9
  textAttr(0)=SADD("topaz.font"+CHR$(0))
  textAttr(1)=height*65536&
  IF FontPtr THEN CloseFont FontPtr
  FontPtr=OpenFont(VARPTR(textAttr(0)))
  IF FontPtr THEN SetFont WINDOW(8),FontPtr
  PRINT
  PRINT " This shows the system font";height;"points high"
NEXT height

WINDOW OUTPUT 1
END
```

Besides setting a particular type size in a window, you can assure accurate placement of text with the PTAB command. The LOCATE command, which is normally used for text placement, moves the text cursor to even character positions. The absolute coordinates of these character positions may vary according to the size of the text font. But PTAB moves the text cursor to an absolute pixel location. Its syntax is

PTAB*(x)*

where *x* is the horizontal coordinate for the text cursor. If you wish to position the text at an absolute vertical coordinate as well, you must use the operating system routine Move, which was demonstrated above in the explanation of the PolyDraw routine in the "Drawing Shapes" section.

SCROLLing

The SCROLL command allows you to "cut" a rectangle out of a window and "paste" it elsewhere in the scene. The syntax for this command is

SCROLL ***rectangle, x_offset, y_offset***

The rectangle value specifies the coordinates of the upper left corner of the area to be scrolled and the lower right corner of that area. It is expressed in the form

(left,top)-(right,bottom)

where *left* and *right* are the horizontal coordinates, and *top* and *bottom*, the vertical coordinates. These are always expressed as absolute coordinates and cannot be expressed relative to the last position drawn, as can the drawing commands.

The x_offset and y_offset values show how far to move the designated area horizontally and vertically. The area to

which you move the rectangle will be covered by it. The area from which the rectangle is moved will be filled in background color.

The following sample program roughly mimics the main action of the arcade game *Space Invaders*, by SCROLLing a group of shapes from side to side and steadily downward.

```
'Box Invaders
SCREEN 1,320,200,2,1     'lo-res, 4-color screen
WINDOW 2,,,0,1           'full-size window
PALETTE 0,0,0,0          'black background
PALETTE 2,1,.2,.2        'red

FOR Row = 0 TO 3         'draw 4 rows
 FOR Column = 0 TO 7 'of 8 boxes
  LINE (Column*30,Row*20)- STEP (20,12),2,bf
NEXT Column, Row

inc=-1: Column=-1

FOR Row= 0 TO 11
 Column = Column-SGN(Column)
 inc = -SGN(inc)
 'move boxes vertically
 SCROLL (Column,Row*10)-(Column+230,Row*10+82),0,10
  FOR Column = 0-80*(inc = -1) TO 80+80*(inc=-1) STEP inc
  'move them horizontally
  SCROLL (Column,Row*10)-(Column+230,Row*10+82),inc,0
NEXT Column,Row

WaitForClick: IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
WINDOW OUTPUT 1
END
```

Saving and Restoring Images (GET and PUT)

Among the most powerful features of the Amiga graphics routines are those that let you manipulate an entire image at a time rather than just lines or points. The first of these allows you to "capture" the image in a rectangular area of the screen in an array, then to redisplay that image elsewhere in the window (or in another window entirely) instantly.

The first step is to draw an image on the screen and store it in an array by using the GET command. The syntax of that command is

GET ***(x1,y1)-(x2,y2), array [(sub1[,sub2...])]***

The two values that must be specified are the rectangle whose image is to be stored and the name of the array in which it will be saved. The area of the rectangle is specified by the coordinate pairs *(x1,y1)* and *(x2,y2)*. The first pair represents the absolute position of the top left corner of the rectangular area, and the second specifies the bottom right corner.

Before we can use an array name to store an image, the size of the array must be declared with the DIM command. Its size must be large enough to hold all of the display data. In determining the size to which the array must be DIMensioned, let's take a look at the format in which the image is stored. If an integer array is used, the first three words store the width, height, and depth of the array. Let's take the case of an image that is 40 dots wide, 20 lines high, and three bit planes deep:

```
a%(0) = 40
a%(1) = 20
a%(2) =  3
```

Since the image data is stored in 16-bit words, the width of the image is rounded up to the next highest multiple of 16 to find the least number of words required to store one line of the image. In this example, each line requires 3 words of data (48 bits) to hold the 40 dots. Since there are 20 lines per bit plane, each bit plane requires 60 words (3 words wide * 20 high) to hold the data. The correspondence of the bit patterns of the data words and the dots that make up the display is the same as that described in the section "Patterned Fills," above. The data for plane 0 is assigned to array elements as follows:

```
a%(3)=line0left      a%(4)=line0middle      a%(5)=line0right
a%(6)=line1left      a%(7)=line1middle      a%(8)=line1right
.                    .                      .
.                    .                      .
.                    .                      .
a%(60)=line19left    a%(61)=line19middle    a%(62)=line19right
```

The same kind of assignment is made for each of the 3 bit planes. Since there are 3 bit planes, a total of 183 words are required (60 words/bit plane * 3 bit planes + 3 format words).

For purposes of your programs, if you use short integer arrays, you can use the following formula to find the size:

arraysize = 3 + INT ((16+x2−x1)/16) * (1+y2−y1) * ***depth***

Use this size to DIMension the array using the statement

DIM a%(*arraysize*)

The GET statement allows you to specify subscripts for the array. This allows you to create multidimensional arrays, with a picture stored in each subscript. For example, if you want to store five images that each require an integer array of 500 words, you may dimension one array for all five images using the statement form

```
DIM a%(500,5)
```

When you fill the array, use the form

```
GET (x1,y1)-(x2,y2),a%(0,0)  'first image
GET (x3,y3)-(x4,y4),a%(0,1)  'second image
GET (x5,y5)-(x6,y6),a%(0,2)  'third image
```

Note that the first subscript always stays at zero, while the second keeps track of the image number.

To redisplay the stored image, you use the PUT statement. The form of this statement is

PUT [STEP] *(x,y), array [(sub1[,sub2...])] [,combination_type]*

Here, you need only specify the coordinates of the top left corner of the image and the name of the array in which it is stored. The coordinates may be specified either as absolute points or as an offset relative to the last point drawn. As with GET, multiple array dimensions may be specified.

What kind of image is drawn when you use the PUT statement depends a lot on the value you choose for *combination_type*. Five types of combinations may be made from the image values stored in the array and the values that are currently displayed on screen. Their names are PSET, PRESET, AND, OR, and XOR. The concepts behind these combinations should not be so strange; PSET and PRESET are graphics commands which we have discussed, and the rest are logical operators whose functioning is discussed in the dictionary of Amiga BASIC words (Chapter 2).

If the PSET combination_type is selected, the entire rectangular area of the image will appear, exactly as it was saved. This includes the background color as well as any foreground colors used. If the PRESET type is chosen, the entire area of the image appears with each color, including the background color, complemented. This means, for example, if the screen is two planes deep, parts of the image that were stored

as color 0 would appear in color 3, parts that were stored as color 1 would appear in color 2, and vice versa. For more information about complementing, see the section "Drawing Modes," above.

Using AND, OR, XOR

The three remaining combination types use the logical operators AND, OR, and XOR (exclusive OR) to combine the pen values of the stored image with those displayed onscreen. In the AND mode, the bits of the image are logically ANDed with those of the display (see the Amiga BASIC dictionary in Chapter 2 for information about the AND operator). The following chart shows all of the possible combinations of one pen color ANDed with another in a four-color display:

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	0
0	2	0
0	3	0
1	1	1
1	2	0
1	3	1
2	2	2
2	3	0
3	3	3

The OR combination mode logically ORs the bits of the image with those of the display. For more information on the OR operator, see the Amiga BASIC dictionary in Chapter 2. The following chart shows all of the possible combinations of one pen color ORed with another in a four-color display:

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	1
0	2	2
0	3	3
1	1	1
1	2	3
1	3	3
2	2	2
2	3	3
3	3	3

The XOR combination mode is the default mode used if no combination_type is specified. That may be because in the XOR mode the image always appears (though its color may vary), and the part of the stored image that was drawn in the background pen never appears. Also, the XOR operator is useful for some animation because it "undoes" its own effects. If you PUT an image once using XOR mode, the image appears, but if you PUT the same image the second time using that mode, the display is restored to its original state before the PUT took place. See if you can figure out why from the following chart, which shows all the possible combinations of one pen color XORed with another in a four-color display:

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	1
0	2	2
0	3	3
1	1	0
1	2	3
1	3	2
2	2	0
2	3	1
3	3	0

The following program graphically demonstrates the various color combinations resulting from use of the different combination_types. Another interesting feature of this program is that the array used by the PUT statement is not created by a corresponding GET statement. Rather than drawing the image, we use the same technique as was demonstrated above to create an area fill pattern. Rows of binary data were laid out one on top of the other to form a picture. This data is then read into the appropriate elements of the image array.

```
DEFINT a-z
WINDOW 1,,(0,0)-(500,180)

DIM man(40)
man(0)=16       'image is 16 bits wide
man(1)=18       'by 18 lines high
man(2)=2        'and 2 bit planes deep

FOR x=3 TO 20
  READ d 'read image data
  man(x)=d 'into the PUT array
NEXT
```

```
FOR row= 1 TO 3     '3 rows
  FOR col=0 TO 4    'of 5 columns each
    LINE (48+100*col,9+50*row)-STEP (20,10),row,bf    'draw a box
  NEXT col

  PUT (50,50*row), man,PSET
  PUT (150,50*row), man,PRESET
  PUT (250,50*row), man,AND
  PUT (350,50*row), man,OR
  PUT (450,50*row), man,XOR
NEXT row

WIDTH 60
LOCATE 5,1    'print heading
PRINT PTAB(40) "PSET" PTAB(130) "PRESET";
PRINT PTAB(240) "AND" PTAB(350) "OR";
PRINT PTAB(440) "XOR"
LOCATE 1,1
REM—18 words of image data
DATA &H07E0, &H0FF0, &H1998, &H1FF8
DATA &H1C38, &H0FF0, &H03C0, &H0FF0
DATA &HFFFF, &HFFFF, &H0FF0, &H0FF0
DATA &H1FF8, &H1FF8, &H1E78, &H1C38
DATA &H7C3E, &H7C3E
```

The next sample program uses the more traditional method of drawing a picture and storing it into the array to be used later by the PUT command. To keep the user from seeing the picture when we first GET it, the PALETTE command is used to change the foreground pen color to the same shade as the background pen color, rendering the drawing "invisible." Also, note that a single two-dimensional array is used to store the image of all six dice. We can display the proper dice face by just changing the second array subscript.

```
DEFINT A-Z
SCREEN 1,320,200,3,1  'lo-res, 8 colors
WINDOW 2,,,8,1        'full-screen window

PALETTE 0,0,0,0   'black background
PALETTE 2,1,0,0   'red foreground
PALETTE 4,0,1,0   'green foreground
PALETTE 5,1,1,1   'white dice spots
PALETTE 3,1,1,1   'white dice spots

GOSUB InitDice 'set up dice image arrays
WIDTH 37: LOCATE 19,6
PRINT "Strike any key to roll again"
RANDOMIZE TIMER 'initialize RND function
y1=80 'top line of dice
```

```
Rolldice:
FOR Change=0 TO 5:FOR Die=1 TO 2  'use two dice, roll each 6 times
  x1=64*Die+40                    'set to left or right die
  Roll(Die)=INT(RND*6)            'pick a random roll
  LINE(x1,y1)-STEP (47,39),Die*2,BF  'blank die
  PUT (x1,y1),Spots(0,Roll(Die))  'draw spots
  SOUND 10000,.001:SOUND 150,0    'make click
NEXT Die, Change

WIDTH 40:LOCATE 9,1
'print numbers above dice
PRINT PTAB(117) Roll(1)+1; PTAB(180)Roll(2)+1

CheckForRoll:
'If user closed the window, end
F WINDOW(8)=0 THEN SCREEN CLOSE 1:WINDOW OUTPUT 1:END
IF INKEY$=" " THEN CheckForRoll ELSE Rolldice

InitDice:
REM This subroutine draws the spots of the dice
REM and then GETs the image data into array SPOTS

DIM Spots(500,7), Roll(2)
PALETTE 1,0,0,0
   'make foreground=background,
   'so spots are invisible

FOR Pair=0 TO 4 STEP 2   'for 3 pairs of dice shapes
  FOR Spot=0 TO 3
    READ x,y
    CIRCLE(x,y),5
    PAINT(x,y)
  NEXT Spot                  'draw two dice
  GET (104,80)-(159,119),Spots(0,Pair)
  GET(168,80)-(223,119),Spots(0,Pair+1)    'read their data
NEXT Pair

PALETTE 1,1,1,1
CLS        'clear screen and make spots white again
RETURN

DATA 178,86,206,112,127,99,127,99
DATA 206,86,178,112,112,86,142,112
DATA 178,99,206,99,142,86,112,112
```

Animated OBJECTs (Vsprites and Bobs)

In addition to the more limited types of animation that you can achieve with the PUT and SCROLL commands, Amiga BASIC contains a number of commands that specifically deal with animated graphics objects. Before we discuss these commands, a short overview of the fundamentals of these objects is in order.

There are two general approaches to animation on microcomputers, and the Amiga supports both of them. The first involves drawing the objects on the normal bitmap screen. To create an animated object on the bitmap screen, you must first save the background image bitmap, draw an object on the screen bitmap, restore the background image, and draw the object elsewhere on the screen. On most systems, this method requires a fair amount of tedious machine language programming. On the Amiga, much of the difficulty is alleviated by the Amiga's hardware and operating system software support. The Amiga hardware includes a sophisticated device known as a blitter (or block image transferrer). This device is able to move whole blocks of bit images at a time, as opposed to using a program to move the image a single byte at a time. Using this hardware device, the operating system supports movable objects called blitter objects, or bobs for short. Because of the operating system support, these objects may be treated as separate entities, even though they are really part of the normal bitmap display.

The other class of animation objects is known as sprites. These are drawn using an entirely separate hardware system from that of the normal bitmap display. The concept of sprites should be familiar to users of the Commodore 64, TI-99/4A, and Atari 400/800 line of computers, all of which support variations of this concept. Moving sprites is much easier than moving bitmap images. Their location at any given time is determined by x and y coordinates stored in a hardware register. The Amiga hardware supports eight hardware sprites. Sprites are always displayed in low-resolution mode, regardless of the current screen resolution, and each is a maximum of 16 pixels wide, though they can be any height. They can contain a maximum of three foreground colors and transparent (background color) each, and each pair of sprites shares a set of color registers (pens).

Many of the hardware limitations of the Amiga's sprites are overcome by the operating system software that controls them. The operating system supports the concept of virtual sprites. Since the display hardware is able to change the display characteristics as each line is displayed, it is possible to change the position of a sprite after it has been displayed and show it again elsewhere lower on the screen. This means that although there are only eight hardware sprites, each of these

may be "reused" many times on a single display, making it possible to have many "virtual" sprites on screen at one time. The only limitation on these virtual sprites is that there cannot be more than four sprites having the same colors in a single horizontal line.

All of the Amiga BASIC OBJECT commands described below work with either bobs or vsprites. There are slight variations on how these commands are carried out, however, due to the inherent differences in the nature of these two kinds of objects. Whether you choose to make an object a bob or a vsprite will depend to some degree on the needs of your program. Some of the differences between the two object types which you should consider are the following:

Size and resolution. Bobs have the same resolution of the screen on which they appear, while vsprites always appear in low resolution. Vsprites can be only 16 pixels wide, while bobs can be virtually any size, so long as you have enough memory to store their shape data.

Number of objects. You can display as many bobs onscreen as you want, but only six vsprites of the same color can appear on a single horizontal line, and only four of different colors.

Speed of motion. Vsprites move quickly, but bobs can be slower, particularly if you are using very large bobs or a lot of them. Generally speaking, the more bobs, the slower they move.

Number and selection of colors. Bobs can use the maximum number of colors available on the screen on which they appear. They are limited to the exact same color selection as any other bitmap object that is drawn on that screen. Vsprites can have only three foreground colors and one background color. But these colors can be completely different from the ones selected for the rest of the screen. The only limitation is that no more than four vsprites of different colors can appear next to each other on the display. Vsprites can therefore be used to add color to a display without using up more bit planes worth of memory. The additional colors are made available by changing the sprites' color registers as vsprites move up and down the screen. Since vsprites use the same color registers as the upper 16 bitmap pen registers, bitmap objects drawn in these colors may change color as the vsprites

move up and down. For this reason, it is not advisable to use vsprites on 32-color screens.

Color priority. Bobs have a selectable priority; you can determine which will be displayed in front of the others. Vsprites always appear in front of bobs.

Hardware system used. Because bobs are part of the normal bitmap display, they fit much better into the windowing environment of Intuition. They move when their windows are moved, they never move outside the borders of their windows into other windows, and they disappear when their windows are covered or closed. None of this can be said for vsprites. Because they use a completely different hardware display system, they don't stay in their windows and will be displayed even after the window is closed. They can also cause color conflicts with the mouse pointer. Because the pointer is actually a sprite, its color registers may be affected when the operating system software changes the sprite color registers. This means that the pointer color may be different in one horizontal part of the screen than in another when you use vsprites.

Creating and Displaying OBJECTs

The first step in creating a movable object is to define its shape. This is done by using the Amiga BASIC program ObjEdit, which is found in the BasicDemos drawer of the *Amiga Extras* disk. This program allows you to draw a bob or vsprite image by using the mouse and then save that image to a disk file. The format of the disk file that the ObjEdit program saves is such that it can be read by your program and used to form a movable object in that image.

Instructions for using the ObjEdit program are found in your Amiga BASIC manual. You should remember, however, that if you edit a sprite using the program supplied, the image of the sprite in the editor will be only half as wide as the vsprite object that appears in your program, because the editor uses the high-resolution mode (640 dots across), while vsprites always appear in low resolution (320 across). Also, unless you alter the program as indicated in the REMarks at the start of the listing, it will edit only four-color objects.

Once you have drawn the shape and saved its image to a file (which for purposes of this example we will name "ImageFile"), the next step is to read that file into a string in

your program. The command lines that your program may use to accomplish this task are

```
OPEN "ImageFile" FOR INPUT as 1
ObjectImage$ = INPUT$(LOF(1),1)
CLOSE 1
```

These statements read the entire image file into one long string. Once the information resides in this string, it may be used by the OBJECT.SHAPE command to create an object having that shape. The syntax for this command is

OBJECT.SHAPE ***object_num, shape_definition_string***

where *object_num* is a number greater than zero that you assign to the object to identify it for future commands, and *shape_definition_string* is the string into which you have read the image file information (here, ObjectImage$). Once you assign the shape data in the string to the object, that string is no longer needed, and you may free up the memory it required by assigning its value to that of the null string (" ").

The OBJECT.SHAPE command also allows you to create a new object which has exactly the same shape as an existing object. The syntax for this form of the command is

OBJECT.SHAPE ***new_object_num, existing_object_num***

where the value *new_object_num* is the identification number of the new object that you are creating, and *existing_object_num* is the identification number of the object whose shape you are using. When you create a new object using this form of the OBJECT.SHAPE command, both objects share that memory area where the image data is shared, and this saves some memory. In all other ways, however, the two objects are separate and may be treated as unique objects. As we will see below, they may even be of different colors.

Once you have assigned a shape to an object or objects, you need only to give the OBJECT.ON command in order to display them. The format for this command is

OBJECT.ON ***[object_num [,object_num...]]***

where the values marked *object_num* are an optional list of the identification numbers of the objects that you wish displayed. If you supply a list of one or more object numbers, only those objects will be displayed. If you use the OBJECT.ON command with no object_num values, all objects

that have been defined using the OBJECT.SHAPE command will be displayed.

To suspend the display of an object temporarily, you may use an OBJECT.OFF statement of the form

OBJECT.OFF *[object_num [,object_num...]]*

where the values marked *object_num* are an optional list of the identification numbers of the objects that you wish to disappear. As above, if you supply a list of one or more object numbers, only those objects will vanish, but if you use the command with no object_num values, all objects that have been defined using the OBJECT.SHAPE command will be turned off.

To disable an object permanently and release all of the memory associated with maintaining its shape and other attributes, you may use the OBJECT.CLOSE statement whose syntax is

OBJECT.CLOSE *[object_num [,object_num...]]*

where the values marked *object_num* are an optional list of the identification numbers of the objects that you wish to disable. As before, if you supply a list of one or more object numbers, only those objects will be closed, but if you use the command with no object_num values, all objects that have been defined using the OBJECT.SHAPE command will be closed.

Setting the OBJECT Color

As mentioned before, vsprites and bobs use different mechanisms for determining the colors in which the object will be displayed. Vsprites use some of the upper 15 color registers and change the contents of those registers as they move. The colors that a vsprite will display are determined by the last six bytes of its ObjEdit file. These six bytes contain three byte pairs representing the three foreground pens. Each pair has the red value in the first byte, and the green and blue packed in the second byte. Each color value is represented by a number from 0 through 15. The green-blue byte contains a number that is the sum of 16 times the green value plus the blue value. In other words,

grnblu = 16 * green + blue

The ObjEdit program always sets the colors of the three

foreground pens to white, black, and orange. If you wish to use other colors for your vsprites, you must alter the ObjEdit program or the file that it produces, or change the color values in the string after it has been read in from the file. Since the last is the simplest approach, it is the one we will use. The following program fragment demonstrates how to change the string. It should appear in your program after the image file has been read into the string and before the OBJECT.SHAPE command that assigns the image in the string to an object. We use the colors black (0,0,0), purple (15,0,15), and cyan (0,15,15), but you can change the red and grnblu values to suit your needs.

```
L=LEN(ObjectImage$)
red1 = 0
grnblu1 = 0
red2 = 15
grnblu2 = 0*16 + 15
red 3 = 0
grnblu3 = 15 * 16 +15
Col$=CHR$(red1) + CHR$(grnblu1)
Col$ = Col$+CHR$(red2) + CHR$(grnblu2)
Col$ = Col$+CHR$(red3) + CHR$(grnblu3)
MID$(ObjectImage$,L-5) = Col$
```

Bobs, on the other hand, take their colors from the same pens as any other normal graphics image on the screen. Which color pen is used to draw the bob is dependent on the bit image data that you create with the ObjEdit program. You can change these colors with the PALETTE statement, but the rest of the graphics images that were drawn with the same pen will change also.

The OBJECT.PLANES command allows you to change the pen used by your bob without changing the the composition of its bit planes. It is not really useful for vsprites, since their color selection works differently, as explained above. The syntax for this command is

OBJECT.PLANES ***object_num [,PlanePick] [,PlaneOnOff]***

PlanePick and *PlaneOnOff* can be thought of as masks that can change the normal order in which the bit planes are displayed. PlanePick is used to determine what bit planes are used for the display. Let's say that you have a two-plane image that uses pens 2 and 3, and you want to display it on a three-plane screen. Normally, the two planes would be dis-

played in planes 0 and 1. But you can set PlanePick to display these as two entirely different planes. You chose these planes by setting PlanePick to the sum of the bit values of the planes in which you wish the object displayed. Each bit value corresponds to 2^*n*, where *n* is the plane number. For example, the bit value of plane 0 is 1 (2^0), the bit value of plane 1 is 2 (2^1), and so forth. The PlanePick value that corresponds to the normal setting of planes 0 and 1 would be 3 (1+2). To display the image in planes 1 and 2, you would set the PlanePick value to 6 (2+4). The part of the image that was created using pen 1 will now be displayed in the color of pen 2, and the part of the image that was created using pen 2 will now be displayed in the color of pen 4. The part of the image that was originally colored in pen 3 (both planes set) will now be shown in the color of pen 6.

The PlaneOnOff value can be used to further enhance the selection of colors. Let's say that in the above example, you wanted to display your object in pen colors 3, 5, and 7 instead of 2, 4, and 6. Using PlanePick alone, this would not be possible, since each of these colors requires that two color planes be set. PlaneOnOff lets you set the color planes that were not chosen in PlaneOnOff. In our example, an image that originally used planes 0 and 1 (pen colors 1, 2, and 3) was changed to use planes 1 and 2 (pen colors 2, 4, and 6). PlaneOnOff lets you set plane 0 as well. If you chose a PlaneOnOff value of 1, which corresponds to plane 0, everywhere that a pixel is set in plane 1 or 2 will also be set in plane 0. This has the effect of adding 1 to the pen values made possible by PlanePick. If PlanePick is set to 6 and PlaneOnOff is set to 1, the parts of the object that were originally drawn in pens 1, 2, and 3 will appear in pen colors 3, 5, and 7 .

When two bobs overlap, there is a question as to which one is drawn "on top of" the other. Left to its own devices, the operating system will make its own determination based on the position of the objects. If you wish one object always to be displayed "in front of" others, you may specify this with the OBJECT.PRIORITY command:

OBJECT.PRIORITY ***object_num, priority***

where *object_num* is the identification number of the object, and the priority value is a number from −32768 to 32767.

Objects with a higher priority number are displayed on top of objects with a lower priority number. Note that this command applies only to bobs; vsprites always appear in front of normal graphics objects like bobs.

Positioning and Moving OBJECTs

You position your movable objects with the OBJECT.X and OBJECT.Y commands. These commands use the syntax

OBJECT.X ***object_num, xposition***
OBJECT.Y ***object_num, yposition***

where *object_num* is the object ID, and the xposition and yposition values are the coordinates of the top left corner of the object. Although vsprites are always displayed in low resolution (320 pixels across), their xposition values are relative to the screen resolution. If vsprites appear on a hi-res screen, their visible range of motion is from −15 to 639. This range is not affected at all by the size of the current Output window, unlike that of bobs, which can be seen only in the visible part of their windows. Regardless of the visible range of the object, the position commands will keep track of an object's position through the range of −32768 to 32767.

You may find that the position of the objects does not change immediately when an OBJECT.X or OBJECT.Y is issued. If no objects are in motion, you may have to wait until a motion command or other command that affects the display occurs.

The position statements also may be used as functions to determine the current x and y position of an object. The syntax for the functions is

xposition = **OBJECT.X** ***(object_num)***
yposition = **OBJECT.Y** ***(object_num)***

where *xposition* and *yposition* represent the current coordinates for the object whose ID number is *object_num.*

Normally a bob will be displayed if positioned anywhere within its window. It is possible to further restrict the visible range of a bob with the OBJECT.CLIP command. The format of this command is

OBJECT.CLIP ***(left,top)-(right, bottom)***

where the first pair of coordinates represent the top left corner of the visible area, and second pair specify the bottom right corner. If you position the bob anywhere outside the specified

area, it will not be displayed. Although clipping does not apply to vsprites, the OBJECT.CLIP command sets the boundaries for the purpose of the collision detection (see below) for both bobs and vsprites.

While it is possible to move your graphics objects by changing their x and y positions, it may require a number of program statements to keep them in motion. Amiga BASIC provides commands which let you move these objects at a constant rate of speed with just a couple of statements. These commands are OBJECT.VX and OBJECT.VY:

OBJECT.VX ***object_num, x_velocity***
OBJECT.VY ***object_num, y_velocity***

The x_velocity and y_velocity values represent the speed of the object in pixels per second. A positive x value moves the object to the right and a positive y value moves the object down. A negative velocity value moves the object in the opposite direction.

After you have set the velocity for an object, you must use the OBJECT.START command to set it in motion:

OBJECT.START ***[object_num [,object_num...]]***

If you specify a list of one or more object_num values, only those objects will start moving. If no object_num value is given, all previously defined objects will move.

To stop an object, you can use the OBJECT.STOP command:

OBJECT.STOP ***[object_num [,object_num...]]***

This command will also apply to specific objects only if a list of object_num values is furnished. Otherwise, all motion is stopped. An object's motion is also stopped when it is made invisible with an OBJECT.OFF command.

Once an object is put into motion, it will keep going until it collides with a border or with another object. Such a collision has the same effect on the object as an OBJECT.STOP command. Therefore, if you want to keep the object in motion, you must periodically check for collisions. You can do this either by using the OBJECT.X and OBJECT.Y functions to check the position of the object, or by using the ON COLLISION command discussed below to change its direction when it reaches the border of the screen. When you have detected a collision, you must start up the object again with an OBJECT.START command.

Like the positioning commands, the velocity commands also can be used as functions to determine the current velocity of an object. The syntax of the functions is

x_velocity = **OBJECT.VX** *(object_num)*
y_velocity = **OBJECT.VY** *(object_num)*

where *x_velocity* and *y_velocity* are the current velocities of *object_num* that are returned by the function.

The following example program brings together a number of the elements discussed above. Since this book cannot transmit the equivalent of an image file created by the ObjEdit, we use a subroutine called InitImage to create the string equivalent of such a file. The image described by the file is that of a bob in the shape of a flying saucer. This image is assigned to two bobs, and the color of one is changed by using the OBJECT.PLANES command. Both bobs are then positioned and set into motion.

```
WINDOW 1,"Unidentified Flying Bobs",(0,0)-(300,186),4

GOSUB InitImage
 'create ShipShape$ from data,
 'instead of reading image file

OBJECT.SHAPE 1, ShipShape$    'create first spaceship
OBJECT.Y 1,50                 'position it vertically
OBJECT.VX 1,60                'give it horizontal motion

OBJECT.SHAPE 2,1              'create second ship
OBJECT.PLANES 2,2,1           ' make it white with orange windows
OBJECT.X 2,150                'position white ship horizontally
OBJECT.Y 2, 180               'and vertically
OBJECT.VY 2,-45               'give it vertical velocity upward

OBJECT.ON                     'display both ships
OBJECT.START                  'start them moving

FOR delay = 1 TO 2300 'kill time while they move
NEXT delay
OBJECT.CLOSE                  'wipe out both objects
END

InitImage:
'Create the string equivalent
'of an ObjEdit image file

FOR x=0 TO 89
 READ d%
 ShipShape$=ShipShape$+CHR$(d%)
```

```
NEXT
RETURN

DATA 0, 0, 0, 0 ,0, 0, 0, 0
DATA 0, 0, 0, 2, 0, 0, 0, 32
DATA 0, 0, 0, 8, 0, 24, 0, 3
DATA 0, 0
' Bit Plane 0
DATA &H00, &H00, &H00, &H00
DATA &H00, &H00, &H00, &H00
DATA &H00, &H00, &H00, &H00
DATA &H03, &HC3, &HC3, &HC0
DATA &H03, &HC3, &HC3, &HC0
DATA &H00, &H00, &H00, &H00
DATA &H00, &H00, &H00, &H00
DATA &H00, &H00, &H00, &H00
'Bit Plane 1
DATA &H00, &H3F, &HFC, &H00
DATA &H03, &HFF, &HFF, &HC0
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &H03, &HFF, &HFF, &HC0
DATA &H00, &H3F, &HFC, &H00
```

Just as the velocity commands allow you to change the position of an object automatically, Amiga BASIC includes acceleration commands that allow you to change the velocity automatically. The format of these commands is

OBJECT.AX ***object_num, x_acceleration_rate***
OBJECT.AY ***object_num, x_acceleration_rate***

where *x_acceleration_rate* and *y_acceleration_rate* represent the velocity to be added to an object's current velocity every second. In other words, it specifies the velocity change in pixels per second per second.

The above caution to watch an object once it is set in motion applies even more strongly when you use the acceleration command. A high rate of acceleration may cause an object to hit a border very quickly. It may even develop "escape velocity," where the object is redrawn at such large intervals that collision checking no longer works. In such a case, it will not stop at the border, but keep right on going and disappear from the display entirely.

These commands may also be used as functions to determine an object's current acceleration rate. The format of the acceleration functions is

x_acceleration_rate* = OBJECT.AX *(object_num)
y_acceleration_rate* = OBJECT.AY *(object_num)

where the function returns the x or y acceleration rate of *object_num.*

Detecting Collisions

When a movable object collides with another object or with one of the borders of the window, Amiga BASIC notes the collision and saves the information about it on a stack. This stack can hold information about only 16 collisions at a time. After the stack is full, BASIC ignores any subsequent collisions.

You can receive several kinds of information about collisions from the COLLISION function. The syntax for the various forms of this command is

***object_num* = COLLISION (0)**
***collision_window* = COLLISION (−1)**
collision_code* = COLLISION *(object_num)

The first form of the function , COLLISION (0), gives the *object_number* of the object that was involved in the collision whose information is the top item on the stack. It leaves the collision information on the stack, where it can be retrieved by a subsequent call of the third form COLLISION *(object_num).*

The second form, COLLISION (−1), identifies the window in which the collision recorded on the top item of the stack occurred. It also leaves the collision information on the stack.

The third form, COLLISION *(object_num)* is the most common. It returns a number, *collision_code,* that identifies what collided with the object in question during the collision recorded by the top entry on the stack. In the process, it also removes the item from the stack to make room for new entries. If you specify the object_num of an object that was not involved in the collision recorded on the top of the stack, the collision_code will be 0, indicating no collision, and you will have lost the chance to find out what happened in that collision. Therefore, if you are unsure which object was involved in the collision recorded on the top of the stack, check it first with COLLISION (0).

Besides zero, indicating no collision, other possible collision_codes include positive numbers, which correspond to the object_num of another object with which the object collided, and negative numbers, which indicate a collision with one of the window borders. The significance of these negative values is

−1 Object collided with top border
−2 Object collided with left border
−3 Object collided with bottom border
−4 Object collided with right border

There is another way to detect collision instead of having your program check the COLLISION function every so often. If you use the ON COLLISION command, BASIC will notify your program every time it detects a collision, and it will cause your program to execute a specified subroutine after the current statement finishes its execution. The format of this command is

ON COLLISION GOSUB ***label***

where *label* is the program label for the subroutine that is to be executed. You can change the subroutine that is to be executed at any time by issuing the ON COLLISION GOSUB command with another label, or disable collision trapping with the statement

ON COLLISION GOSUB 0

Like other event trapping commands, the ON COLLISION statement will not actually direct the program to your subroutine when a collision happens until you give the command

COLLISION ON

It will however still place collision event information in its stack, so when the COLLISION ON command comes, the program will be directed to the specified subroutine once for each collision event that has been stored. After you have given the COLLISION ON command, you may suspend event trapping with the statement

COLLISION STOP

which will stop it until the next COLLISION ON statement. To end collision trapping entirely, use the statement

COLLISION OFF

Normally, Amiga BASIC records collisions between every object, and between objects and the window borders. In some cases, however, you may not want to take any action if certain objects collide with each other or with the border. You may not want BASIC to take any notice of these collisions at all. You can prevent the detection of certain collisions with the OBJECT.HIT command, which takes the form

OBJECT.HIT ***object_num [MeMask] [,HitMask]***

where *MeMask* and *HitMask* are values whose bit patterns determine which type of object will collide. Think of MeMask as a number that defines the collision type of this object, and HitMask as a number that describes the collision type of the object with which this object will collide. If you logically AND the MeMask of one object with the HitMask of another, a collision will be detected only if the result is not a zero. In addition, if the HitMask of an object is an odd number (has a one as the least significant bit), it will collide with borders.

For example, let's take the following four objects:

Object_num	MeMask	HitMask	Collides With
1	0010 (2)	1101 (13)	obj2, obj3, borders
2	0100 (4)	1010 (10)	obj1, obj3
3	1000 (8)	0110 (6)	obj1, obj2
4	0010 (2)	0001 (1)	Borders only

Object 1 has a HitMask that indicates it collides with all object types except those that have the same MeMask as it does. Its HitMask value is 13, which produces a nonzero result when ANDed with either the MeMask of object 2 (4) or the MeMask of object 3 (8). But 13 and the MeMask of object 4 (2) equal 0. Therefore, object 1 collides with both objects 2 and 3, but not object 4. Since its HitMask is odd, it also collides with borders.

Objects 2 and 3 have HitMasks with each other's MeMask bit set in addition to that of object 1. They collide with each other and object 1, but not with the border, since both are even numbers.

Object 4 has a HitMask with only the least significant bit set. It has zeros in the bit places represented by the MeMasks of all the other objects. Therefore, it collides only with the borders.

In the above example, none of the objects had HitMasks indicating that they could collide with another object, unless

that other object also had a HitMask indicating that it collided with the first. Since the position of the objects will determine whether object 1 collides with object 2, or object 2 collides with object 1, it's a good practice to make sure that the HitMasks of each are set so that both collide with each other or neither collides with each other. Otherwise, the collision of the two objects will be reported on some occasions, but not others.

The following sample program is adapted from the Demo program which appears on the *Extras* disk. It demonstrates many of the commands explained in this section. It creates the shape of a flying-saucer vsprite from data and changes the data so that a second vsprite is shown in different colors. It moves and accelerates these vsprites, and uses collision trapping to "bounce" them off the borders. It uses collision masking to make sure that only border collisions are detected, not collisions between the ships. The demo stops after a certain number of bounces to make sure that the ships do not reach "escape velocity" and disappear off the screen.

```
DEFINT a-z
WINDOW 1,"Bouncing Spaceships",(0,0)-(300,186),4

GOSUB Initialize 'set up sprites
WHILE Running
 SLEEP    'do something only when the sprites collide
WEND

OBJECT.CLOSE      'release all objects
PALETTE 0,0,.3,.6 'screen back to blue
END

Bounce:
 T = T+1:IF T = 25 THEN Running = 0
 s = COLLISION(0) 'which object collided?
 IF s = 0 THEN Exeunt 'no more on stack—exit
 c = COLLISION(s) 'what did it collide with?
 vx = OBJECT.VX(s)
 vy = OBJECT.VY(s)
 IF (c=-1 AND vy < 0) OR (c=-3 AND vy > 0) THEN
  'object hit top or bottom border
  OBJECT.VY s,-vy
 ELSEIF (c=-2 AND vx < 0) OR (c=-4 AND vx > 0) THEN
  'object hit left or right border
  OBJECT.VX s,-vx
 END IF
 GOTO Bounce      'check for more collisions on stack
```

```
Exeunt:
  OBJECT.START    'make it go again
  RETURN

Initialize:
  Running = 1
  PALETTE 0,0,0,0
  GOSUB InitImage
  'Create ShipShape$ from data
  'rather than reading image file

  OBJECT.SHAPE 1, ShipShape$
  OBJECT.Y 1,20
  OBJECT.VX 1,10
  OBJECT.VY 1,7
  OBJECT.AY 1,1
  OBJECT.AX 1,2

  L=LEN(ShipShape$)     'change color data of vsprite
  red1=0
  grnblu1=0             ' 0,0,0 = black
  red2=15
  grnblu2 = 0*16+15     ' 15,0,15 = purple
  red3=0
  grnblu3=15*16+3       ' 0, 15, 15 = aqua
  Col$=CHR$(red1)+CHR$(grnblu1)
  Col$=Col$+CHR$(red2)+CHR$(grnblu2)
  Col$=Col$+CHR$(red3)+CHR$(grnblu3)
  MID$(ShipShape$,L-5)=Col$

  OBJECT.SHAPE 2, ShipShape$    'create a saucer with new
                                colors

  OBJECT.Y 2,30
  OBJECT.VX 2,2
  OBJECT.VY 2,2
  OBJECT.AY 2,2
  OBJECT.AX 2,2

  OBJECT.ON                     'make them visible
  OBJECT.START                  'start them moving
  OBJECT.CLIP (0,0)-(275,184)   'set borders
  OBJECT.HIT 1,2,1
  OBJECT.HIT 2,2,1              'only collide with borders
  ON COLLISION GOSUB Bounce     'set collision trapping
  COLLISION ON                  'turn it on

  RETURN
```

```
InitImage:
FOR x=0 TO 63
  READ d%
  ShipShape$=ShipShape$+CHR$(d%)
NEXT
RETURN

DATA 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 2, 0, 0, 0, 16
DATA 0, 0, 0, 8, 0, 25, 0, 3
DATA 0, 0, 0, 0, 0, 0, 0, 0
DATA 25, 152, 25, 152, 0, 0, 0, 0
DATA 0, 0

'Sprite Image Data
'2 bytes wide by
'8 lines high

DATA &H07, &HE0
DATA &H1F, &HF8
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &H1F, &HF8
DATA &H07, &HE0

  'Sprite colors
  'RGB values are held in two
  'hex bytes—0R and GB

DATA &H00, &H00
DATA &H0F, &H00
DATA &H0F, &HF0
```

Chapter 6

Programming Amiga Sound

Philip I. Nelson

Programming Amiga Sound

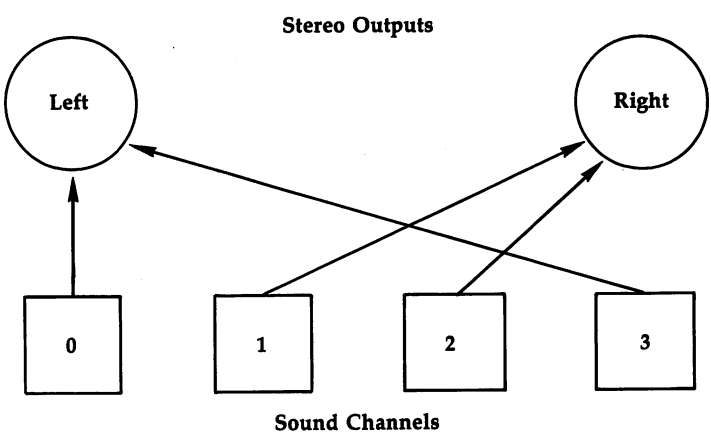
Philip I. Nelson

The Amiga offers more sonic power and flexibility than any other home computer. As you'll see in the course of this chapter, you can make a great many different sounds (including speech) on the Amiga without being a programming wizard.

We'll begin with a discussion of how Amiga BASIC creates speech, including simple text-to-English translation and more sophisticated phoneme-based speech with which the Amiga can speak in any language. Next, we'll look at the SOUND and WAVE commands, demonstrating how to use them for music and sound effects. The chapter concludes with a discussion of advanced sound features and a short example program written in machine language.

Before you begin to program sound and music on the Amiga, you must understand how its sound channels are connected. As you may have heard, the Amiga has four sound channels and two stereo outputs. Each channel can generate a sound (or even multiple tones) independent of the others. Here's how the four sound channels are connected to the two stereo outputs:

Figure 6-1. Amiga Sound Connections



As you can see, the four channels are numbered 0–3. Channels 0 and 3 are connected to the left stereo output, and channels 1 and 2 connect to the right stereo output. It's important to keep these relationships in mind since they are wired into the computer and you therefore can't change them by programming. If your monitor has only one speaker, you'll want to make sure that all the sounds you create come out that speaker. If you have two speakers connected to your Amiga, you can send sounds, music, and speech through one speaker at a time or through both at once for complex stereo effects. But now let's make the Amiga talk.

Speech

One of the most revolutionary features of the Amiga is its ability to speak. To see how easy it is to create synthesized speech, enter this statement in the BASIC Output window:

```
SAY TRANSLATE$("My Amiga can talk.")
```

If this is the first SAY statement you have executed during this session, the Amiga must first load a program called the *narrator device* from disk. When that's done, the computer pronounces the phrase clearly. This is the simplest method of speech generation, and it works in both immediate mode and program mode. The SAY command tells the computer that you want it to speak, and the TRANSLATE$ function supplies the word or phrase you want to say.

If you use the SAY command, but the narrator program isn't available on the current disk, the Amiga displays a requester box that tells you to insert the appropriate disk. Unfortunately, if this occurs after you have opened a custom Output window, the requester box will appear in the original Output window and may thus be invisible. So, in any program with speech, it's a good idea to place a SAY statement at the beginning, before the program opens any custom windows, to allow for this eventuality.

TRANSLATE$ requires that you enclose information (in this case, a string constant or variable) in parentheses. The example shown above used the string constant "My Amiga can talk." But string variables work just as well and are often more useful than constants. For instance, enter these statements in the Output window:

```
Talk$="My "+CHR$(65)+"miga can talk"+CHR$(46)
SAY TRANSLATE$(Talk$)
SAY TRANSLATE$(MID$(Talk$,10,4)+LEFT$(Talk$,9)+MID$(Talk$,13,5))
```

The TRANSLATE$ function accepts nearly anything that would be appropriate to an ordinary PRINT statement: It lets you concatenate (combine) a string variable from several different elements or manipulate the variable with string functions like LEFT$ and MID$.

What exactly does TRANSLATE$ do? To find out, enter these statements from the Output window:

```
Talk$="What does this function do?"
PRINT TRANSLATE$(Talk$)
SAY TRANSLATE$(Talk$)
```

The Amiga prints these characters:

```
WHAHT DAH4Z DHIHS FAH4NXKSHUN DUW.
```

This combination of letters and numbers may look peculiar at first, but it all makes sense to the narrator device. When you execute a SAY TRANSLATE$ statement, TRANSLATE$ converts the string of English text into a string of *phonemes*—the basic building blocks of spoken English—which the narrator device needs in order to pronounce the phrase. We'll take a closer look at phonemes later in this chapter. For now, note that the SAY command always requires phonemes. The TRANSLATE$ function will translate English text into phonemes. But you can also skip the translation and supply the phonemes yourself. These statements do exactly the same thing:

```
SAY TRANSLATE$("hi there.")
SAY "/HAY4 DHEH1R."
```

If you don't care to learn about phonemes, you can let the Amiga perform the translation for you with TRANSLATE$. No matter what sort of string you provide, the computer does its best to convert it into clear, intelligible speech. Considering the complexity of the task, the TRANSLATE$ function does an excellent job. Program 6-1 lets you type in any string, hear it spoken, see the resulting phoneme string, and experiment with any voice parameter.

Program 6-1. Speech Experimenter

```
'Speech Experimenter.
'Set Preferences for 80 columns.

GOSUB Setup

Getit:
  LOCATE 1,1
  PRINT CHR$(124) 'Phony cursor.
  Check=MOUSE(0)  'Read left mouse button.
  Key$=INKEY$     'Read keyboard.
  'Nothing happening.
  IF Key$="" AND Check=0 THEN Getit
  'Repeat previous phrase if RETURN pressed.
  IF Key$=CHR$(13) THEN GOSUB Sayit:GOTO Getit
  'Mouse button was pressed.
  IF Check<>0 THEN Change
  'Key other than RETURN was pressed.
  'Get remainder of input line.
  LOCATE 1,1:PRINT Key$;
  LINE INPUT Text$
  Talk$=Key$+Text$
  IF Talk$<>"quit" AND Talk$<>"Quit" THEN
    'Not done yet.
    GOSUB Sayit
    GOTO Getit
  END IF
  'Graceful exit.
  Talk$="OK. Bye-bye."
  GOSUB Sayit
  'Close custom window
  WINDOW CLOSE 2
  END

Sayit:
  'Display and pronounce current phrase
  'with current voice parameters.
  LOCATE 1,1
  PRINT Talk$
  LOCATE 3,1
  PRINT SPACE$(70)
  LOCATE 3,1
  PRINT TRANSLATE$(Talk$)
  SAY TRANSLATE$(Talk$),Voice%
  LOCATE 1,1
  PRINT SPACE$(70)
  RETURN
```

```
Change:
  x=MOUSE(1) 'Mouse pointer horizontal coordinate.
  y=MOUSE(2) 'Mouse pointer vertical coordinate.
  'Is mouse pointer outside selection zone?
  IF y<32 OR y>1Ø3 OR x>1Ø8 THEN Getit
  'Left mouse button was pressed in selection zone.
  'Highlight current selection.
  This=INT(y/8)-3
  PUT (Ø,Vpos(This)),Box
  'Input new value.
  GOSUB Lokate
  PRINT SPACE$(7)
  GOSUB Lokate
  PRINT CHR$(32);
  INPUT Number
  'Check for illegal input.
  IF Number<Low(This) OR Number>High(This) THEN
    Temp$=Talk$
    Talk$="I can't use that number."
    GOSUB Sayit
    Talk$=Temp$
    GOTO Nogood
  END IF
  'Input is legal, make it current.
  Voice%(This-1)=Number
  Now(This)=Number
  'Branch here for illegal input.
  Nogood:
  GOSUB Lokate
  PRINT SPACE$(8)
  GOSUB Lokate
  PRINT Now(This)
  'Erase highlight box.
  PUT (Ø,Vpos(This)),Box
  'Prevent false mouse/keyboard readings.
  Check=Ø
  LOCATE 5,1
  FOR j=1 TO 9
    Garbage=MOUSE(Ø)
    'Reprint prompts in case of Redo from start err
or.
    PRINT Prompt$(j)
    x$=INKEY$
  NEXT
  GOTO Getit

Lokate:
  LOCATE This+4,16
  RETURN
```

```
Setup:
  'Say something before opening custom window.
  SAY TRANSLATE$("hi there.")
  PALETTE 0,.04,.03,.02
  PALETTE 1,.9,.93,.94
  PALETTE 2,.05,.6,.5
  PALETTE 3,.85,.85,.02
  WINDOW 2, "Speech Experimenter"
  'Create highlight box.
  LINE (0,0)-(110,7),,bf
  DIM Box(100)
  GET (0,0)-(110,7),Box
  PUT (0,0),Box
  'Create highlight box.
  LINE (0,0)-(110,7),,bf
  DIM Box(100)
  GET (0,0)-(110,7),Box
  PUT (0,0),Box
  'Draw screen and initialize arrays.
  PRINT CHR$(124)
  PRINT "TRANSLATE$ Conversion:"
  PRINT:PRINT
  Vert=32
  FOR j=1 TO 9
    READ Prompt$(j),Low(j),Now(j),High(j)
    PRINT Prompt$(j);Now(j),Low(j);"-";High(j)
    'Voice array must use elements 0-8
    Voice%(j-1)=Now(j)
    Vpos(j)=Vert
    Vert=Vert+8
  NEXT
  LOCATE 15,17
  PRINT CHR$(94);SPACE$(12);CHR$(94)
  PRINT SPACE$(16);"Current";SPACE$(6);"Legal"
  PRINT SPACE$(16);"Value";SPACE$(8);"Range"
  LINE (220,28)-(340,140),,b
  DATA "Pitch           ",65,110,320
  DATA "Inflection      ",0,0,1
  DATA "Rate            ",40,150,400
  DATA "Gender          ",0,0,1
  DATA "Frequency       ",5000,22200,28000
  DATA "Volume          ",0,64,64
  DATA "Channel         ",0,10,11
  DATA "Synch mode      ",0,0,1
  DATA "Synch control   ",0,0,2
  'Welcoming message
  Talk$="Please type what you want me to say."
  GOSUB Sayit
  Talk$="Type quit when you want to stop."
  GOSUB Sayit
  RETURN
```

Program 6-1 begins by speaking and printing some brief instructions. Note that every sentence is displayed both in English and in phoneme form. If you enter a new phrase, the program pronounces it as soon as you press RETURN and displays the phrase in phoneme form as well. You may capitalize letters if you like, but TRANSLATE$ doesn't distinguish between lowercase and uppercase. One of the easiest ways to learn how to use phonemes is to enter words with this program and observe how TRANSLATE$ converts them into phoneme strings. Later in this chapter, you'll find a program that lets you construct words and phrases directly from phonemes.

Also on the screen, just below the phoneme display, you'll see all of the current voice array settings, along with the allowable range of values for each element. To change a voice setting, press the left mouse button once on the element you want to change, then enter a value within the range shown to the right. After changing a voice array element, you can enter a new phrase or repeat the last phrase (press RETURN without entering anything).

The TRANSLATE$ function has several built-in features that may not be apparent at first. For example, run Program 6-1 and enter these phrases:

oc
ok

The narrator device pronounces *oc* phonetically, but says *ok* as *okay,* just as we do in normal speech. The phoneme string for *oc* is AA4K, but for *ok* TRANSLATE$ automatically substitutes the phonemes OWKEY3. Here are some other examples to try:

pi = 3.14159265
"Don't quote me."
#2
2+2=4
4&4=8
$43.21
@6
7%
...
....
5 is > 1.
2 is < 9.
.1
.e
e.
e.1
1.e

The narrator device pronounces numbers one at a time: For example, 51 is said as "five-one," not "fifty-one." The plus (+) and equal (=) signs are pronounced explicitly as well. However, the minus sign (−) is treated as a hyphen: The

phrase 5–5 is pronounced as "five-five," not as "five minus five." The period (.) is pronounced as "point" only when it comes right before a number. In other cases a period is either ignored or treated as an end-of-sentence marker that causes a dropping inflection in the narrator's voice. Table 6-1 contains all the special combinations pronounced by TRANSLATE$.

Table 6-1. TRANSLATE$ Special Features

Symbol	Pronounced As
.	Point (before a number)
<	Less than
>	Greater than
/	Slash
=	Equals
"	Quote/unquote
#	Number
$	Dollar
%	Percent
&	And
^	Caret
~	Tilde
|	Or
...	And so on

Several punctuation marks are not pronounced explicitly by TRANSLATE$. The hyphen, which normally connects two related words or symbols, has the same effect as a blank space. The period, colon, and semicolon are all treated as end-of-phrase markers: They cause a pause and a dropping of pitch in the narrator's voice. Let's try a few examples with Program 6-1 to see exactly what these symbols do. Enter this phrase:

hi there you all

If you don't include an end-of-phrase symbol at the end of a string, the Amiga pronounces it without dropping the pitch of its voice. Now add a period at the end.

hi there you all.

With most sentences, this will sound more natural. TRANSLATE$ treats the exclamation point and question mark exactly the same: No extra emphasis is added for an exclamation point, and the question mark does not cause the pitch to rise.

hi there you all.
hi there you all!
hi there you all?

If you want to say something in the form of a question with TRANSLATE$, try ending the sentence without an end-of-phrase symbol. In many cases this sounds more "questioning" than a sentence that ends in the normal way. (As explained below, the question mark and certain other characters have a different effect when you create speech directly from phonemes without using TRANSLATE$.)

If you add a comma to the middle of this sentence, it sounds even more natural:

hi there you all.
hi there, you all.

Note that the comma causes a shorter pause and a less drastic lowering of pitch, appropriate for separating closely related phrases within a sentence. The colon and semicolon have the same effect as a period:

hi there. you all.
hi there; you all.
hi there: you all.

Changing speech qualities. It's quite easy to give your Amiga's voice different personalities by changing its pitch, speech rate, and other factors. This is done by adding an array reference to the end of a SAY statement. This voice array must be an integer array and must contain nine elements. Each element of the voice array controls a different aspect of the narrator's voice (Table 6-2).

Table 6-2. Voice Array Elements

Element	What It Controls	Range
1	Pitch	65–320
2	Inflection	0–1
3	Rate	40–400
4	Gender	0–1
5	Sampling frequency	5000–28000
6	Volume	0–64
7	Channel assignment	0–11
8	Synchronization mode	0–1
9	Synchronization control	0–2

Program 6-1 lets you experiment with different voice array elements. To change the pitch of the narrator's voice, for instance, simply move the mouse pointer onto the word *Pitch* in the lower part of the display, then press the selection button once. Now you can enter a new pitch value. If you enter a value outside the range displayed to the right, the program speaks and displays an error message and cancels the operation. After making the change, you can repeat the most recent phrase or enter a new one as usual.

A SAY TRANSLATE$ command can take two different forms. If you are satisfied with the default voice settings, use the simpler form shown here:

SAY TRANSLATE$(text string$)

It's necessary to create a voice array only when you want to use custom voice settings. To include custom voice features, you must use this syntax:

SAY TRANSLATE$(text string$),voice array%

Adding a voice array reference to the end of a SAY TRANSLATE$ statement tells the Amiga to speak the phrase defined by the text string in the way you have specified in the voice array. Of course, you can use any legal Amiga BASIC variable name for the array: Way%, DOG%, and v% are all acceptable.

When a voice array is included, each element of the array must be within the ranges in Table 6-2. Values outside those limits cause an "Illegal Function Call" error in Amiga BASIC. If this error occurs when using a voice array, check each value to make sure it's within the allowable range. Let's look at each voice array element in turn.

Pitch (element 1). The first element of the voice array sets the pitch of the narrator's voice—whether it sounds high or low. The default pitch setting of 110 is considered to be average for a male voice. Pitch values can range from 65 (very low) to 320 (very high).

Inflection (element 2). This element of the voice array takes a value of either 0 (inflected speech) or 1 (monotone). The default setting of 0 causes the narrator's voice to rise and fall naturally as it speaks each word. If you set the inflection value to 0, the narrator's voice becomes more robotlike.

Rate (element 3). The rate value controls how fast the narrator talks, a very important aspect of speech. The normal set-

ting is 150, but you may specify other values from 40 (very slow) to 400 (fastest). The speech rate that you choose depends somewhat on the comprehensibility of what's being said and the type of person who's listening. If you're writing a program for small children, for instance, you'll probably want to slow it down. To avoid irritating delays in some situations, you may want the narrator to speak fast: In a program written for sophisticated users, prompts and warning messages can be spoken quickly. Much faster rates are acceptable when you display a message as well as speak it since the user then has two means of comprehending the information.

Gender (element 4). The default setting for this parameter is 0, which selects a "male" voice. Substitute a 1 to switch to a "female" voice. While it's difficult to describe the difference precisely, the effect of selecting a female voice is much like routing the male voice through a highpass filter: The voice's fuller, low-frequency tones are suppressed, making it sound lighter and somewhat more nasal. The female voice may or may not sound truly female to your ears: To simulate a particular voice more exactly, try increasing the pitch and/or the sampling frequency in addition to this element.

Sampling frequency (element 5). This parameter controls the quality of the voice and, indirectly, its pitch. Strictly speaking, it determines how often (in terms of hertz, or cycles per second) the narrator device samples the basic voice waveform. In practical terms, the higher the sampling frequency, the more inflected the voice sounds. You may use values from 5000 (very low) to 28000, with the default being 22200, quite high in the available range. Values in the lower part of the range sound much the same—like giants in low-budget fantasy films—while very high sampling values make the narrator sound childish or munchkinlike. When defining a custom voice of your own, you'll probably want to experiment with different sampling frequencies as well as different pitches, since both parameters affect the overall pitch of the narrator's voice and the overall contour of the sound.

Volume (element 6). This will almost always be 64, the maximum and also the default setting. Lower values decrease the volume, down to a setting of 0 which is inaudible. The volume setting for speech relates directly to the volume chosen in SOUND statements (see below). A SAY volume setting of 64 is just as loud as a SOUND setting of 64, but SOUND

statements can range in volume all the way up to 255, which can drown out the loudest speech. If you're mixing speech with other sounds, you'll want to confine all SOUND statements to a volume near 64 to match the loudest speech that SAY can produce.

Channel assignment (element 7). This value controls which of the four available sound channels receives speech output. Since the Amiga mixes its four sound channels into two stereo outputs, this setting also determines whether speech comes out the left or right stereo speaker, or both. The default channel setting of 10 causes speech to issue from any available left/right pair of channels. This helps insure that all speech is audible with a monophonic hookup, no matter which speaker is connected. In a stereo hookup it always produces stereo speech unless fewer than two channels are free. Table 6-3 outlines all of the voice channel assignments.

Table 6-3. Voice Channel Assignments

Value	Channel
0	0 (left speaker only)
1	1 (right speaker only)
2	2 (right speaker only)
3	3 (left speaker only)
4	0 and 1 (left and right speakers)
5	0 and 2 (left and right speakers)
6	1 and 3 (left and right speakers)
7	2 and 3 (left and right speakers)
8	Either 0 or 3 (left speaker)
9	Either 1 or 2 (right speaker)
10	Any free left/right combination (left and right)
11	0 or 1 or 2 or 3 (right or left speaker)

Most of these voice channel assignments are important only when you need to produce speech at the same time that other sound effects (including other speech) are in progress. In other cases the default setting works just fine. Channel assignments 0–7 force speech output to the channel or channels that you designate. You may choose a value 0–3 to send the speech through any single channel, or select a value 4–7 to route the voice output through a designated *pair* of channels.

The remaining channel assignments are less specific and

leave part of the decision making up to the Amiga. If you choose a value of 8, the computer sends the speech through either channel 0 or 3, to the left speaker. A channel assignment of 9 sends speech through either channel 1 or 2, to the right speaker. (Note that assignments 8 and 9 both send output through only one channel.) The default setting of 10 chooses any free left/right channel combination. This is convenient, but assumes that at least two channels are free. In the busiest situations, a setting of 11 may be the safest choice, since it selects any single channel that happens to be free at the moment.

Keep in mind when assigning voice channels that not every Amiga owner will have a stereo hookup. If you are writing programs for a large audience, you should do everything possible to insure that everyone using your program will be able to hear the sounds you have taken so much time to create. This may mean sacrificing some stereo effects, limiting the number of simultaneous sounds, or providing a stereo/mono option to the user.

Synchronization mode (element 8). The normal setting here is 0, which causes Amiga BASIC to wait until the current SAY command is finished before processing any further Amiga BASIC commands. This mode is called *asynchronous*, since the second command waits for the completion of the first rather than proceeding synchronously (at the same time). That's fine if you want to freeze your program until the speech has finished. For example, if an error occurs, you may want to display a warning box that requires a yes or no response from the user. In this case, asynchronous mode pauses program execution until the message has been said and thereby emphasizes the warning.

In other cases you'll want to minimize the delay between the onset of speech and the execution of subsequent program statements by selecting *synchronous* mode (use a setting of 1). In synchronous mode the narrator permits the execution of subsequent Amiga BASIC statements as soon as it has digested the SAY string rather than waiting until everything has been said. While the subsequent statements are in progress, the speech continues "in the background." Synchronous speech is very useful when you want to minimize delays.

For reasons explained earlier, it's desirable to SAY a short message at the very beginning of any program that includes

speech. And, if this is done in synchronous mode, other setup tasks can be performed while the message is in progress.

You can easily observe the difference between synchronous and asynchronous speech with Program 6-1. Enter a long phrase of six to eight words, then press RETURN twice to repeat the phrase twice in succession. In asynchronous mode (0), the phrase is not displayed a second time until the first speech is finished. In synchronous mode (1), the redisplay is completed before the first phrase ends.

Synchronization control (element 9). This parameter is important primarily when you are dealing with synchronous speech. It controls what happens when BASIC encounters a second SAY statement before it completes a preceding SAY statement. When you use the default setting of 0, the narrator politely waits for the first SAY statement to finish before it says anything more. Setting this parameter to 1 selects *cancel* mode: The Amiga immediately cancels the preceding SAY command. If you test this mode with Program 6-1, no speech is heard, regardless of which synchronization mode you select. A synchronization control setting of 2 activates *interruption* mode: In this mode a second SAY statement will interrupt any preceding SAY statement that's still in progress (if the preceding SAY command hasn't yet begun or has already finished, no difference is noticeable). This mode might be useful in a multiplayer game, where you would want to permit one player to interrupt another's speech. Table 6-4 illustrates the effect of different synchronization combinations.

Table 6-4. Voice Synchronization

Sync Mode	Sync Control	Result
0	0	Asynchronous speech
1	0	Synchronous speech
0	1	Cancel mode
1	1	Cancel mode
0	2	Asynchronous speech
1	2	Interruption mode

Phoneme-based speech. The Amiga's narrator device program can produce speech only when supplied with phonemes—the building blocks of human speech. In the preced-

ing section, we saw how Amiga BASIC's TRANSLATE$ function performs that conversion automatically. In many cases a simple SAY TRANSLATE$ statement will be all you'll need to create effective, comprehensible speech. But English is an extremely complex language, with many conflicting pronunciation rules and a number of words absorbed from other languages.

Since the Amiga's narrator device program must be small enough to fit into the computer's memory (along with Amiga BASIC and the current program text), it can't possibly account for every peculiarity of spoken English. Thus, while the narrator does an extraordinary job of pronouncing most words comprehensibly, there are also many exceptions. For instance, run Program 6-1 and enter the following phrase, taken from act 2, scene 2 of Shakespeare's *Hamlet.*

the satirical rogue says that old men have grey beards

TRANSLATE$ has no trouble pronouncing common words like *the* and *have.* Because such words occur so often, it's critical that they be pronounced correctly and the TRANSLATE$ function is customized to handle a number of them as special cases. For instance, the words *cave* and *pave* are pronounced with a long *a* vowel, so the word *have* must be treated as an exception to a rule. Similarly, words like *phe* or *dhe,* or any single-syllable word ending with *e,* are pronounced with a long *e,* demonstrating that *the* is also a special case.

Certain parts of words are also singled out for special handling. For example, if you enter the word *conversion* with Program 6-1, the syllable *sion* is translated correctly into the phoneme string SHUN. But if you enter *sion* as a separate word, it's converted to a different phoneme string (SIH4UN).

Less common words and syllables must be translated letter-by-letter, according to general rules that work well most of the time, but fail in a significant number of cases. Attempting our phrase from Shakespeare, the narrator badly mispronounces *satirical* and *rogue. Satirical* is translated into SAE4TIHRIHKUL, and *rogue* becomes RAA4G, neither of which sounds correct.

In many cases the simplest solution is to deliberately misspell the word in a more phonetic form. For example, *sohterikll* and *roag* will generate the phoneme strings SAATEH3RIHKL

and ROW4G, both of which sound much better. The emphasis in *satirical* is shifted from the first to the second syllable, and the vowel sounds in both words are closer to the correct pronunciation. In simple cases, where speech is not a major factor in the program or you just want to get a message across, deliberate misspelling often does the trick. Program 6-1 makes it easy to experiment by trial and error: When you find a spelling that sounds right, write it down and use it in your program.

Deliberate misspelling has two additional advantages over using phoneme strings. First, a string you supply to TRANSLATE$ will almost always be shorter than an equivalent phoneme string. Most phonemes are two characters long, even where the English equivalent is just one character. That may not seem like a big difference, but it can be a major factor when a lot of speech is involved. Second, it's often desirable to use a single subroutine that handles all the speech in a given program. Using phoneme strings in some cases and English text in others means that you'll need two speech routines—one for text and another for phonemes.

Nevertheless, phoneme-based speech does permit complete control over the narrator device, resulting in precision pronunciations. If you find TRANSLATE$ unsatisfactory, phonemes will usually produce an acceptable result. In fact, you can even make the narrator speak in other languages, limited largely, however, to languages which, like English, derive from Indo-European roots. There are many other languages—particularly African and Asian tongues—which require sounds that are difficult or impossible to produce with the Amiga narrator's phoneme set.

One difficulty when using phonemes is that the SAY command is very fussy about them. Every phoneme must be in uppercase, and only certain character combinations are permissible. If you accidentally include a lowercase character or violate the narrator's internal rules, Amiga BASIC signals an "Illegal Function Call" error; so it's important to know what phoneme combinations are allowed.

Appendix H of the Amiga BASIC manual contains a good deal of information about creating phonetic speech from Amiga BASIC. But the best way to learn about phonemes is simply to use them and observe the results. If you've never used phonemes before, Program 6-1 provides a good introduc-

tion. By entering different phrases and seeing what phoneme strings TRANSLATE$ provides, you can learn which phonemes the computer chooses to represent certain sounds. If you'd like to try constructing words directly from phonemes, Program 6-2 simplifies the process.

Program 6-2. Phoneme Builder

```
'Phoneme Builder
'Set Preferences for 80 column text.

GOSUB Setup

Readmouse:
x$=INKEY$
IF x$=CHR$(32) THEN GOSUB Gotaspace
IF MOUSE(0)<>1 THEN Readmouse
Newx=MOUSE(1)
Newy=MOUSE(2)
'Scan all boxes.
FOR j=1 TO 72
  IF Newx=>Upx(j) AND Newx=<Downx(j) AND Newy=>Upy(
j) AND Newy=<Downy(j) THEN
    Newx=Upx(j):Newy=Upy(j)
    IF Oldx=Newx AND Oldy=Newy AND j<>71 THEN Skipo
ut
    IF j>69 THEN Specials
    IF Small=1 THEN PUT (Oldx,Oldy),Box
    Small=1
    Oldx=Newx:Oldy=Newy
    PUT (Oldx,Oldy),Box
    MOUSE OFF
    GOSUB Sayphoneme
    MOUSE ON
  END IF
Skipout:
NEXT
GOTO Readmouse

Sayphoneme:
'Pronounce the current phoneme.
Current$=Phoneme$(j)
'SAYing either RX or LX by itself causes a crash.
IF Current$<>"RX" AND Current$<>"LX" THEN SAY Curre
nt$,Voice%
Small=1
RETURN
```

```
Specials:
'Add phoneme to word, say word or clear it.
IF Small=1 THEN Small=0:PUT (Oldx,Oldy),Box
Oldx=Newx:Oldy=Newy
PUT (Oldx,Oldy),Bigbox
ON j-69 GOSUB Addit, Sayit, Newit
PUT (Oldx,Oldy),Bigbox
GOTO Skipout

Gotaspace:
PUT (Upx(70),Upy(70)),Bigbox
Current$=CHR$(32)
PUT (Upx(70),Upy(70)),Bigbox
'Fall thru to Addit.

Addit:
Word$=Word$+Current$
LOCATE 16,7
PRINT Word$;CHR$(60)
'Fall thru to Sayit .

Sayit:
IF Word$<>"RX" AND Word$<>"LX" THEN SAY Word$,Voice
%
RETURN

Newit:
Word$=""
LOCATE 16,7
PRINT CHR$(60);SPACE$(60)
RETURN

Setup:
DIM Voice%(8)
FOR j=0 TO 8
READ Voice%(j)
NEXT
DATA 110,0,150,0,22200,64,10,1,0
Welcome$="WEH4LKAHM TUW DHAX FOH7NIYM BIH2LDER."
PRINT Welcome$
SAY Welcome$,Voice%
Makescreen:
CLS
PALETTE 0,.04,.03,.02
PALETTE 1,.9,.93,.94
PALETTE 2,.05,.6,.5
PALETTE 3,.85,.85,.02
LINE (0,0)-(23,12),3,bf
DIM Box(30)
GET (0,0)-(23,12),Box
```

```
PUT (0,0),Box,XOR
LINE (340,101)-(394,113),3,bf
DIM Bigbox(100)
GET (340,101)-(394,113),Bigbox
PUT (340,101),Bigbox
DIM Phoneme$(69)
x=1
FOR k=2 TO 12 STEP 3
  FOR j=2 TO 16
    READ a$
    LOCATE k,j*4
    PRINT a$
    Phoneme$(x)=a$
    x=x+1
  NEXT
NEXT
FOR j=61 TO 69
  READ Phoneme$(j)
NEXT
x=1
FOR j=7 TO 39 STEP 4
  LOCATE 14,j
  PRINT x
  x=x+1
NEXT
LOCATE 14,45:PRINT "Add";SPACE$(5);"Say";SPACE$(5);
"New"
DIM Upx(72),Upy(72)
DIM Downx(72),Downy(72)
x=1
FOR k=5 TO 78 STEP 24
  FOR j=52 TO 500 STEP 32
    LINE (j,k)-(j+23,k+12),,b
    Upx(x)=j
    Upy(x)=k
    Downx(x)=j+23
    Downy(x)=k+12
    x=x+1
  NEXT
NEXT
x=61
FOR j=52 TO 330 STEP 32
  Upx(x)=j:Upy(x)=101
  Downx(x)=j+23:Downy(x)=113
  LINE (j,101)-(j+23,113),,b
  x=x+1
NEXT
FOR j=340 TO 468 STEP 64
  LINE (j,101)-(j+54,113),,b
  Upx(x)=j:Upy(x)=101
```

```
  Downx(x)=j+54:Downy(x)=113
  x=x+1
NEXT
LOCATE 16,7
PRINT CHR$(60)
RETURN

'Phonemes must all be uppercase.
DATA AA,AE,AH,AO,AW,AX,AY
DATA B,CH,/C,D,DH,DX,EH,ER,EY
DATA F,G,/H
DATA IH,IL,IM,IN,IX,IY
DATA J,K,L,LX
DATA M,N,NX,OH,OW,OY,P
DATA Q,QX,R,RX
DATA S,SH,T,TH
DATA UH,UL,UM,UN,UW
DATA V,W,Y,Z,ZH
DATA ".","?","-",",","(",")"
DATA "1","2","3","4","5","6","7","8","9"
```

When you run Program 6-2, it delivers a welcome, then displays a screen containing all of the Amiga phonemes. To hear what a phoneme sounds like, simply move the mouse pointer into the desired box, then press the selection button. Once you have selected a phoneme, you have two choices: You can either move the pointer to the Add box and press the selection button to add the phoneme to the current phrase, or move to another phoneme box to hear how that one sounds. To add a blank space between words, simply press the space bar. Note that certain phonemes, such as punctuation symbols or emphasis numbers, don't have any audible effect until they're combined with other phonemes.

After you add a phoneme, the Amiga will pronounce the entire current phrase. By clicking the selection button on the Say or New box, you can either repeat the current phrase or erase it to begin a new phrase. Table 6-5 shows all of the phonemes used by the narrator device.

AmigaDOS speech capabilities. AmigaDOS 1.1 includes two different versions of the SAY command—an interactive command for experimentation and a direct command that works something like SAY TRANSLATE$ in Amiga BASIC. While voice synthesis might seem somewhat out of place in the spartan DOS environment, speech can be as useful here as

anywhere else. Instead of simply ECHOing prompts or warning messages to the screen, why not SAY them as well?

The AmigaDOS version of SAY differs from the Amiga BASIC command in three important respects. First, AmigaDOS does not accept phonemes as input: The string that you supply to SAY must be English text, which the narrator converts into phonemes (just as if you had used TRANSLATE$ in Amiga BASIC). Second, only four of the nine voice parameters (gender, inflection, rate, and pitch) can be changed. However, since AmigaDOS permits you to insert parameter-changing codes anywhere in a text string, it's a bit easier to change the narrator's voice "on the fly" than in Amiga BASIC. Finally, custom voice settings do not persist from one AmigaDOS command line to the next: If you omit voice codes from a SAY command, AmigaDOS reverts to the default voice settings.

For a fun application, why not have your Amiga greet you by name when it boots up? This can be done by adding a SAY command to the Startup-Sequence file that executes when you boot a system disk. Startup-Sequence is ordinarily found in the S subdirectory and can be edited like any other ASCII text file (use a word processor or the AmigaDOS ED command; be sure to resave Startup-Sequence as ordinary ASCII text, without any formatting characters or special control codes). Hearing a friendly word or two helps liven the otherwise tedious process of waiting for the disk to boot. Spoken messages can also emphasize unusual features of a disk, as demonstrated by this modified version of Startup-Sequence:

```
ECHO "Hi, Melissa"
SAY -f -p175 hi, melissa. this is machine language disk #4.
ECHO "ML Work Disk #4."
SAY -f -p175 relax while i set up the ramdisk for you.
ECHO "Copying command directory to RAMdisk..."
LoadWB
endcli > nil:
```

Since this example is just for demonstration, we've left out the lines that would copy disk-resident commands into the RAMdisk (see Chapter 4). Appendix B contains more information about the AmigaDOS version of SAY.

Table 6-5. Amiga Phonemes

Phoneme	Type	Sounds Like
AA	Vowel	Pop
AE	Vowel	Fan
AH	Vowel	Fun
AO	Vowel	Walk
AW	Dipthong	Flower
AX	Vowel	Abound
AY	Dipthong	Ride
B	Consonant	Bank
CH	Consonant	Chap
/C	Consonant	Loch
D	Consonant	Dare
DH	Consonant	The
DX	Special	Kitty
EH	Vowel	Men
ER	Vowel	Word
EY	Dipthong	Same
IH	Vowel	Sip
IX	Vowel	Rapid
IY	Vowel	Sleep
F	Consonant	Fat
G	Consonant	Goon
/H	Consonant	Hat
IL	Contraction	Same as IXL
IM	Contraction	Same as IXM
IN	Contraction	Same as IXN
J	Consonant	Joke, genius
K	Consonant	Cap
L	Consonant	Parallel
LX*	Special	Ball
M	Consonant	Map
N	Consonant	Nanny
NX	Consonant	Tang
OH	Vowel	Fort
OY	Dipthong	Join
OW	Dipthong	Tow
P	Consonant	Pip
Q	Special	Mitten
QX	Special	(Silent pause)
R	Consonant	Rap
RX*	Special	Far
S	Consonant	See
SH	Consonant	Fish

Phoneme	Type	Sounds Like
T	Consonant	Top
TH	Consonant	Path
UH	Consonant	Book
UW	Dipthong	Too
UL	Contraction	Same as AXL
UM	Contraction	Same as AXM
UN	Contraction	Same as AXN
V	Consonant	Van
W	Consonant	Trowel
Y	Consonant	You
Z	Consonant	Enclosure
.	Special	End sentence
?	Special	End question
-	Special	Connector
,	Special	End phrase
( )	Special	Enclose phrase
1–9	Special	Emphasize syllable

* Amiga BASIC 1.1 crashes the computer when you SAY the phonemes LX or RX in isolation. To prevent this, Program 6-2 pronounces them only when they're combined with at least one other phoneme.

SOUND and WAVE in Amiga BASIC

In addition to speech, Amiga BASIC makes it quite easy to create music and a variety of sound effects. This is done with SOUND, which takes the following general form:

SOUND frequency, duration, *volume, channel*

The simplest SOUND effects require only two numbers, which set the sound's frequency and duration. For instance, this statement plays an A-natural note for one second:

```
SOUND 440, 18.2
```

Let's look at each of the SOUND parameters.

Frequency. The first number in every SOUND statement sets the frequency (pitch) of the sound; it can be any number from 20 through 15000. This represents an actual frequency value (measured in *hertz*, or cycles per second). For instance, the example above generates a 440 hertz tone—the A above middle C—exactly the same tuning pitch used by orchestras around the world. Doubling the frequency produces an A one octave higher, and halving it produces the next lower A. By way of comparison, the topmost note on a piano (a C-natural) has a frequency of about 4186 hertz.

In the equal tempered scale—the most common scale for music—each octave is divided into 12 half-steps, or *semitones*, and the difference in frequency between one semitone and another is always the same. Thus, for conventional music you can generate the required frequency values with straightforward arithmetic. Each half-step is about 1.059463 (the twelfth root of 2) times higher in frequency than the last. If A-natural equals 440 hertz, the next higher semitone (A#) is about 466.1637 (440 * 1.059463) hertz, and so on. Table 6-6 contains typical frequency values for an equal tempered musical scale.

Table 6-6. Equal Tempered Scale

Note	Octave	Frequency
A	0	27.500
A	1	55.000
A	2	110.000
A	3	220.000
A#	3	233.068
B	3	246.928
C	4	261.624
C#	4	277.200
D	4	293.656
D#	4	311.124
E	4	329.648
F	4	349.228
F#	4	370.040
G	4	392.040
G#	4	415.316
A	4	440.000
A	5	880.000
A	6	1760.000
A	7	3520.000

Notice that the relationship between frequency and pitch is roughly exponential. At the low end of the scale, your ears perceive an octave's difference between frequencies of 110 and 220. As you move up the scale, much bigger changes in frequency are needed to produce the same effect. You can generate a similar scale in Amiga BASIC with the following routine, which includes frequencies for the 88 notes on a piano (few musical pieces require notes outside this range).

Appendix C contains a table of the frequency numbers produced by this routine.

```
DIM Frequency#(88)
x#=LOG(27.5#)/LOG(2#)
FOR j=1 TO 88
  Frequency(j)=2^(x#+j/12#)
NEXT
```

Other formulas can be used to much the same effect. Of course, you're not restricted to using this or any other rational scale, and for certain types of music a slightly different formula may work better. For instance, you might follow the same general scheme, but base the scale on C rather than A. The overall results would be similar—each C would have a frequency twice that of the next lower C—but the frequencies of the semitones between octaves would be different. Of course, since there aren't any conventions to govern ray guns or screams, you can use much coarser methods for sound effects. In fact, a bit of dissonance often heightens the impact of an arcade-type sound.

Duration. Every SOUND statement must also specify a duration value within the range 0–77. A duration value of 18.2 makes the sound last for one second. A duration of 36.4 lasts two seconds, and the maximum duration of 77 lasts roughly 4½ seconds. The only way to make longer sounds is to execute successive SOUND commands; if this is done without intervening delays, you won't notice any breaks. Very small durations are useful, too. Try this:

```
FOR k=0 TO 3
  FOR j=1000 TO 6000 STEP 300
    SOUND j, .1
  NEXT
NEXT
```

Volume. To specify a certain volume with SOUND, supply a value from 0 (silence) through 255 (loudest) after the duration, separating the two values with a comma. This parameter is optional: If you don't specify a volume, SOUND automatically uses a value near the middle of the range. By altering the volume, you can make a sound fade in and out:

```
FOR j=0 to 255 STEP 5
  SOUND 440,1,j
NEXT
```

```
FOR j=255 TO 0 STEP -5
  SOUND 440,1,j
NEXT
```

It's difficult to overemphasize the importance of dynamic volume changes in music. Few experiences are more boring than listening to a song that has no volume dynamics whatsoever. If you mix voice commands with SOUNDs, you'll soon notice that the loudest speech is no louder than a SOUND statement played at a volume of 64. To prevent the speech from being overpowered, you may want to keep simultaneous SOUND statements near the same volume level. Volume settings also become important when you use more than one custom waveform at once (see below), since certain waveforms are much louder than others. One feature that's missing from Amiga BASIC is an easy means for generating a sound envelope: When you begin a SOUND statement, it goes immediately from no sound to the specified volume. And every SOUND terminates just as quickly. As we'll see later in this chapter, this limitation can be overcome via machine language.

Channel. The final parameter in a SOUND statement, also optional, selects one of the four available sound channels. These values correspond directly to the channel assignments illustrated in Figure 6-1. Channels 0 and 3 send output to the left stereo speaker, and channels 1 and 2 are connected to the right speaker. If you don't specify a channel, SOUND uses only channel 0 (left speaker). The easiest way to make multivoice music is to play notes through different channels at the same time. For instance, this short program plays a four-note chord in the key of C.

```
SOUND 523.25,20,255,0
SOUND 659.26,20,255,1
SOUND 783.99,20,255,2
SOUND 1046.5,20,255,3
```

In this example every note has the same duration (20) and volume (255), but each has a different pitch and channel assignment, so all four notes of the chord begin and end together. Like the SAY command, SOUND has the ability to play "in the background" while the computer goes on to perform subsequent Amiga BASIC statements. To illustrate, add these statements to the end of the last program and run it again:

```
FOR j=1 TO 10
  PRINT "Done."
NEXT
```

As soon as the sounds begin, the computer proceeds to print *Done* ten times, finishing while the sounds are still audible. If the Amiga lacked this capability, the program would freeze while SOUND statements were in progress, making it much more difficult to integrate music and sound with other program events.

The Amiga's four channels can play simultaneously, but the same is not true of a single channel when SOUND is used. To illustrate, change the 3 in the last SOUND statement to 0 and rerun the program. This time, the Amiga plays a three-note chord followed by a single note. The computer can't start a new note for any given channel while *that channel* is still busy producing a previous note.

Synchronizing multiple SOUND commands. You may wonder how the Amiga keeps track of which notes to play when several SOUND statements are involved at once. It's done by setting up a *queue*, which works like the waiting line in a grocery store. The first SOUND statement in the program is the first to be played: Each subsequent SOUND statement is "lined up" behind the first and played immediately (if it uses a free channel) or as soon as it's appropriate (if it uses a channel that's already busy). In most cases this process is transparent, since the computer sets up the queue very quickly. However, if many SOUND statements are involved, the processing takes more time. Run the following program and notice the pause before the PRINT command executes.

```
FOR j=1 TO 20
  FOR k=0 TO 3
    SOUND 220*(k+1),20,64,k
  NEXT
NEXT
PRINT "Done."
```

Before it can execute the PRINT statement, the computer has to digest 80 SOUND statements, causing a noticeable delay. However, the earliest SOUND statements still begin immediately and continue in the background while later ones are being processed.

In other cases—when playing complex, multivoice music, for instance—you may want all SOUNDs to wait until just the

right moment. This can be done with a SOUND WAIT command. As soon as it encounters SOUND WAIT in a program, the Amiga queues up subsequent SOUND commands, but does not play them until it encounters a SOUND RESUME command. At that point the floodgates open and the sound queue is processed in the usual first-in, first-out manner. The most obvious use for SOUND WAIT and SOUND RESUME is to synchronize notes in multivoice music that has notes of assorted durations.

When the Amiga performs SOUND WAIT, it stores the information it will later need in a special memory area called the *system heap*. Each SOUND command performed during a SOUND WAIT interval uses up a certain amount of heap space. Trying to perform too many SOUND statements before a SOUND RESUME occurs may cause a rather nasty error. To illustrate, add this statement to the beginning of the last example program:

```
SOUND WAIT
```

In Amiga BASIC 1.1, all of the BASIC windows are pushed down on the screen to make room for a red message box which informs you that the heap is exhausted. (Press the left mouse button to recover control. If you delete the SOUND WAIT statement from the program and run it again, you'll see that no harm was done.)

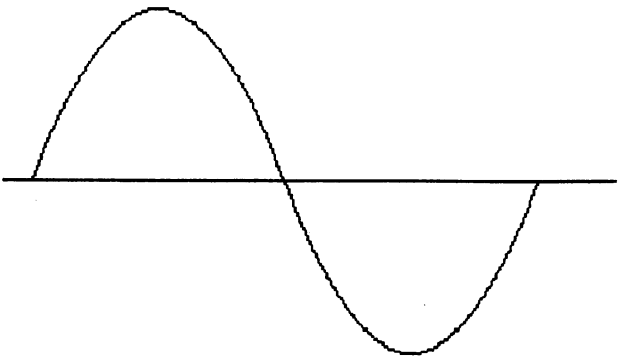
Unless you specify otherwise, the system heap is 1024 bytes long; this special zone is also used by WAVE, LIBRARY, WINDOW, and SCREEN statements. If a program runs out of heap space, you must either modify it to use less heap memory or use CLEAR to allocate extra RAM.

Warning sounds. Apart from music and arcade-type effects, SOUND is handy for emphasizing an error condition. Amiga BASIC itself executes a BEEP command when untrapped errors occur. But this does not occur when you trap errors with ON ERROR and an appropriate subroutine (the Handler routine in Program 6-3 shows how this can be done). If you have a specific error-trapping routine, you can accentuate the impact of an error message by including a BEEP or a more interesting SOUND of your own.

The WAVE command. Every sound consists of a particular pattern of vibrations, which repeats in time. This pattern, called a waveform, gives each sound its own distinctive

character. The simplest waveform is the sine wave, pictured in Figure 6-2.

Figure 6-2. Sine Waveform



In order to produce any sounds at all, the Amiga must have a waveform sample stored somewhere in memory. If you don't specify a waveform of your own, the Amiga automatically uses a sine waveform, which gives every sound a soft, full tone. The WAVE command lets you assign a waveform to any of the four sound channels.

For the sake of simplicity, waveforms are usually portrayed in the two-dimensional form in Figure 6-2. In reality, a sound wave radiates through the air, much like the ripples that appear when you drop a stone in a still pond. The shape in Figure 6-2 actually represents a cross section of such a wave.

The term *sine* is derived from the sine function in trigonometry. Conveniently, SIN is also an Amiga BASIC function: You can assign a sine waveform to any channel with this statement, replacing *channel number* with a value from 0 through 3 (although the SIN function usually requires parentheses, you should *not* include parentheses when using SIN with WAVE).

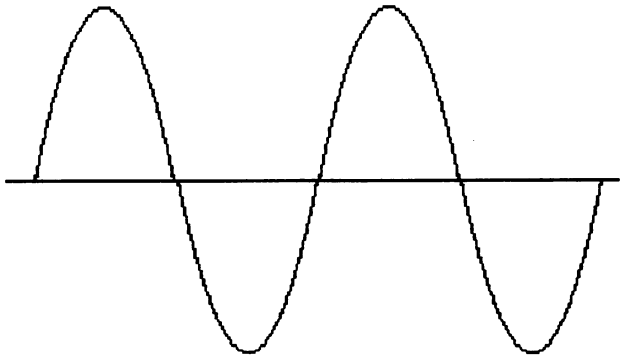
WAVE channel number, SIN

A pure sine wave is based on a mathematical ideal. The waveforms produced in nature (for instance, those made by conventional musical instruments) are far more complex and sound richer because they include many harmonic frequencies in addition to the fundamental frequency produced by the

wave's basic shape. Nevertheless, because sine waves can be created with simple formulas, they're used extensively in electronic sound synthesis. More complex waveforms can be derived by summing together the data for two or more different sine waves.

Every waveform can be defined in terms of two characteristics: amplitude and frequency. Amplitude (the vertical dimension in Figure 6-2) is roughly equivalent to volume: The greater the amplitude range, the louder the waveform sounds. Frequency (the horizontal dimension in Figure 6-2) determines how quickly the waveform repeats within a given time period and thus determines its pitch. If you increase the frequency, the waveform repeats more rapidly and sounds higher in pitch. Figure 6-3 is another sine wave with a higher frequency than the first.

Figure 6-3. Higher Frequency Sine Wave



Creating custom waveforms. To use a custom waveform in Amiga BASIC, you must supply the computer with a new waveform sample in the form of a numeric integer array. The array must have at least 256 elements (larger arrays are permitted, but Amiga BASIC ignores everything above the two hundred fifty-sixth element). Once this is done, you can assign the waveform to a channel with a WAVE command. The following code creates a simple sine wave and assigns it to channel 0:

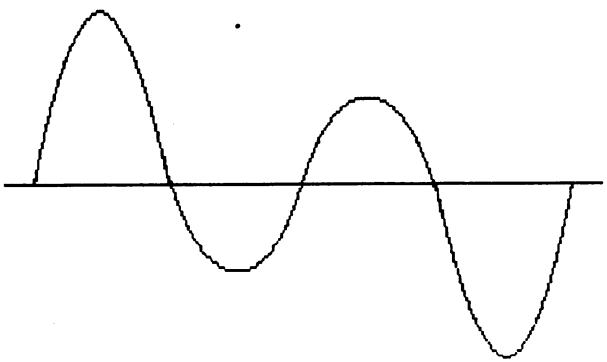
```
DIM Wav%(255)
p# = 2 * 3.14159265# / 256
FOR j=0 to 255
  Wav%(j)=127 * SIN(j*p#)
NEXT
WAVE 0, Wav%
SOUND 220,10,255,0
```

If you run this program, the note sounds unexceptional since this waveform is very similar to the pure sine wave in Figure 6-2 (which the Amiga uses by default). Substitute this line for the fourth line and rerun the program:

```
Wav%(j)=63 * (SIN(j*p#)+SIN(j*2*p#))
```

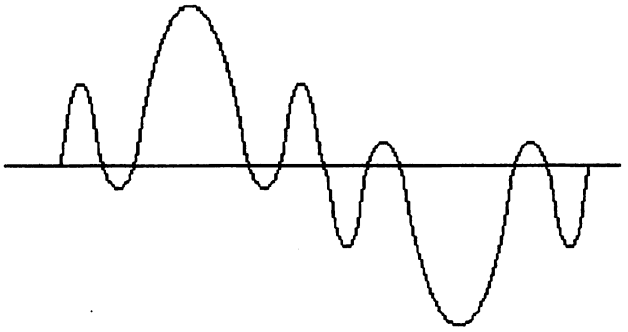
An interesting change has taken place. Now the single SOUND command seems to be producing two notes at once. The lower note is the one we heard before; the second is an octave higher and somewhat quieter. What you've done is sum (combine) the information from two different waveforms. The resulting single waveform generates two distinct frequencies (note that it was necessary to change the 127 to 63 to keep the values within range). Figure 6-4 illustrates the waveform created by this formula.

Figure 6-4. Summed Waveform



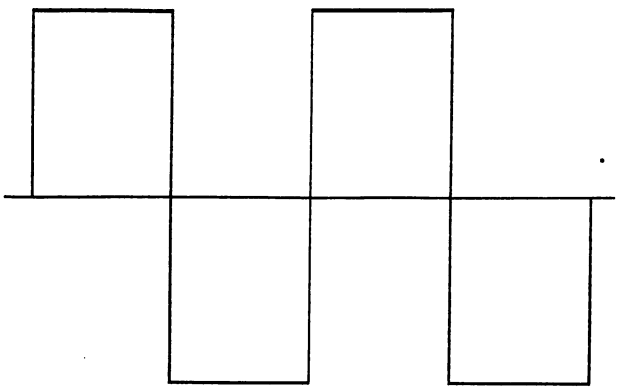
Although the Amiga has only four sound channels, its ability to use custom waveforms makes it possible to generate more than one note for each channel. To take the process even further, change the 2 in the fourth line to a 5, and run the program again. Figure 6-5 shows you this more complex waveform.

Figure 6-5. Complex Summed Waveform



Now the two-note effect is even more pronounced, since this formula creates a dissonant combination of frequencies. Figure 6-6 illustrates a very different sort of waveform, the rectangular wave, often called a square wave.

Figure 6-6. Rectangular Waveform



Rectangular waves produce a louder, somewhat harsher sound than sine waves. To hear what they sound like, enter and run this program:

```
DIM Wav%(255)
FOR j=0 TO 255
  Wav%(j)=127
  IF j>127 then Wav%(j) = -Wav%(j)
NEXT
WAVE 0, Wav%
SOUND 220,10,255,0
```

The rectangular waveform generated by this program is much more powerful than the sine wave, particularly at low frequencies. To reduce its amplitude, change the 127 in the third line of the program to some lower value. Notice what a simple formula we used to generate this wave. The first 127 array elements are set to 127, and the rest are set to −127, proof that you can create a variety of different sounds without being especially mathematical. The triangle and sawtooth waveforms, pictured in Figures 6-7 and 6-8, are also used frequently in electronic sound synthesis.

Figure 6-7. Triangle Waveform

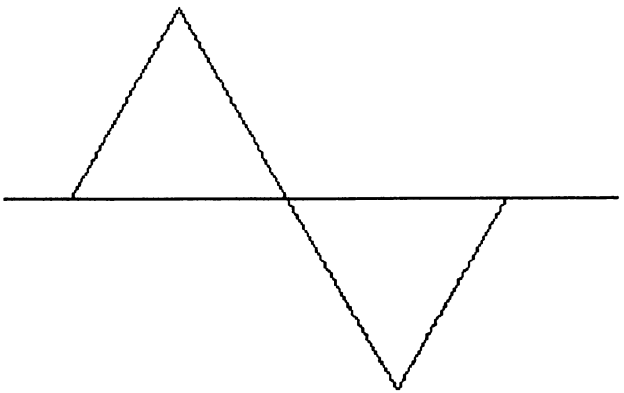
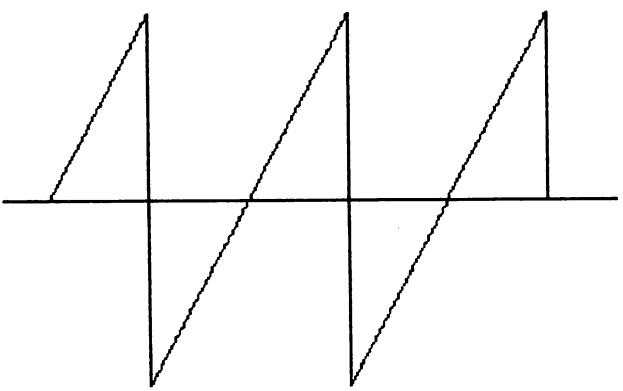


Figure 6-8. Sawtooth Waveform



The triangle wave is soft in tone and also very rich in harmonic frequencies. A sawtooth wave is louder and can sound harsh or buzzy.

Waveform Builder Program. If you're not familiar with waveforms and trigonometric formulas, you may find it hard to visualize what's happening when you create a waveform sample. Program 6-3 simplifies the process, letting you draw a waveform directly on the Amiga's hi-res screen and hear what the waveform sounds like at different frequencies.

Program 6-3. Waveform Builder

```
'Waveform Builder
'Set Preferences to 80 columns

ON ERROR GOTO Handler
GOSUB Setup

Checkmouse:
  x$=INKEY$
  IF x$<>"" THEN Keys
  Check=MOUSE(0)
  IF Check=0 THEN Checkmouse
  x=MOUSE(5):y=MOUSE(6)
  IF x>255 OR y=0 OR y>128 THEN Checkmouse
  'Redraw waveform
  IF y<=64 THEN Nuw=128-2*y
  IF y>64 THEN  Nuw=-INT((y-64)*2)
  PRESET (x,Vert(x))
  PSET (x,y)
  Vert(x)=y
  Wav(x)=Nuw
  GOTO Checkmouse

Keys:
  IF x$=CHR$(13) THEN GOSUB Soundit
  IF x$=CHR$(29) OR x$=CHR$(31) THEN GOSUB Lower
  IF x$=CHR$(28) OR x$=CHR$(30) THEN GOSUB Higher
  IF UCASE$(x$)="C" THEN GOSUB Clerit
  IF UCASE$(x$)="L" THEN GOSUB Lodit
  IF UCASE$(x$)="O" THEN GOSUB Wavit
  IF UCASE$(x$)="S" THEN GOSUB Savit
  GOTO Checkmouse

Lower:
  Freq=Freq-5*(ASC(x$) MOD 28)
  IF Freq<0 THEN Freq=0
  GOSUB Display
  RETURN
```

```
Higher:
  Freq=Freq+5*(ASC(x$) MOD 27)
  IF Freq>32767 THEN Freq=32767

Display:
  LOCATE 13,57
  PRINT Freq;SPACE$(4)
  RETURN

Linit:
  LINE (0,64)-(256,64)
  RETURN

Clerit:
  GOSUB Whatname
  PRINT "Empty"
  FOR j=0 TO 255
    Wav(j)=0
    PUT (j,0),Blank,AND
  NEXT
  Filename$=""
  GOSUB Linit
  RETURN

Whatname:
  LOCATE 11,57
  PRINT SPACE$(20)
  LOCATE 11,57
  RETURN

Soundit:
  'If you don't hear any sound on a one-speaker sys
tem,
  'Change the 1's to 0's in the next two lines.
  WAVE 1,Wav
  SOUND Freq,10,255,1
  RETURN

Savit:
  GOSUB Namit
  IF LEN(Filename$)=0 THEN Skipit
  INPUT "Filename";Filename$
  OPEN Filename$ FOR OUTPUT AS #1
  FOR j=0 TO 255
    PRINT #1, Wav(j)
  NEXT
  CLOSE 1
  RETURN
```

```
Namit:
  GOSUB Spacit
  INPUT "Filename";Filename$
  GOSUB Spacit
  RETURN

Lodit:
  GOSUB Namit
  IF LEN(Filename$)=Ø THEN Skipit
  OPEN Filename$ FOR INPUT AS #1
  FOR j=Ø TO 255
    INPUT #1, x
    Wav(j)=x
  NEXT
  CLOSE 1
  Flag=1-Flag
  GOSUB Wavit
  Flag=1-Flag

Skipit:
  RETURN

Setup:
  DEFINT a-z
  PALETTE Ø,.Ø4,.Ø3,.Ø2
  PALETTE 1,.9,.93,.94
  PALETTE 2,.Ø5,.6,.5
  PALETTE 3,.85,.85,.Ø2
  DIM Blank(3ØØ),Wav(255),Savit(255),Vert(255)
  GET (Ø,Ø)-(Ø,128),Blank
  Freq=22Ø
  Flag=Ø
  LOCATE 2,45
  PRINT "C...........Clear waveform"
  PRINT TAB(45);"L...........Load waveform"
  PRINT TAB(45);"O...........Original waveform"
  PRINT TAB(45);"S...........Save waveform"
  PRINT TAB(45);"RETURN......Hear sound"
  PRINT TAB(45);"CRSR U/R....Higher frequency"
  PRINT TAB(45);"CRSR D/L....Lower frequency
  PRINT
  PRINT
  PRINT TAB(45);"Waveform....Empty"
  PRINT
  PRINT TAB(45);"Frequency...";Freq
  LINE (256,Ø)-(256,129)
  LINE (Ø,129)-(256,129)
  LOCATE 1,34
  PRINT 127
  LOCATE 9,34
```

```
  PRINT Ø
  LOCATE 17,34
  PRINT -128
  LOCATE 8,4
  PRINT "Building a waveform..."
  p# = 2 * 3.14159265# / 256
  FOR j=Ø TO 255
    Savit(j)=31*(SIN(j*p#)+SIN(2*j*p#)+SIN(3*j*p#)+
SIN(4*j*p#))
  NEXT j

Wavit:
  GOSUB Whatname
  IF Flag=Ø THEN Filename$="Original"
  GOSUB Whatname
  PRINT Filename$
  FOR j=Ø TO 255
    PUT (j,Ø),Blank,AND
    IF Flag=Ø THEN Wav(j)=Savit(j)
    IF Wav(j)=>Ø THEN Vertpos=64-Wav(j)/2
    IF Wav(j)<Ø THEN Vertpos=64+(-Wav(j)/2)
    PSET (j,Vertpos)
    Vert(j)=Vertpos
  NEXT
  GOSUB Linit
  GOSUB Soundit
  RETURN

Spacit:
  LOCATE 15,45
  PRINT SPACE$(3Ø)
  LOCATE 15,45
  RETURN

Handler:
  GOSUB Spacit
  Er$=""
  IF ERR=53 THEN Er$="File not found":GOTO Erout
  IF ERR=7Ø THEN Er$="Disk write-protected":GOTO Er
out
  PRINT "Error";ERR
  CLOSE
  RESUME Checkmouse

Erout:
  GOSUB Spacit
  PRINT Er$
  RESUME Checkmouse
```

Program 6-3 begins with a short pause while it constructs a complex, sine-based waveform. When this is done, it displays the waveform on the screen and waits for your commands. If you press RETURN, the program plays a note at the frequency displayed to the lower right. To increase or decrease the frequency value by 5, press the cursor up or cursor down key, respectively. Press cursor right or cursor left to raise or lower the frequency by 15.

To redraw the waveform, simply move the mouse pointer into the waveform display box, press the selection button, and begin dragging the mouse with the button held down. The old waveform is replaced by the new one as you draw. Since the Amiga lets you move the mouse pointer much faster than Amiga BASIC can respond, slow drawing motions give the best results. (The horizontal line in the center of the drawing field is just there for reference; since the center line isn't part of the waveform, it doesn't matter if you erase parts of it when drawing.) You can play notes at any stage during the drawing process. Simply release the mouse button and press RETURN.

Once you create a waveform you like, press S. The Amiga prompts you to enter a filename, then saves the waveform data in a disk file of that name. If you want to start with a clean slate, press C to erase the current waveform and begin a new one. You can also restore the original waveform (press O) or reload any waveform that you previously saved. If you press L, the Amiga prompts you to enter a filename, then loads the data from disk and displays the new waveform.

If you examine the Setup: portion of Program 6-3, you'll see that it creates the default waveform with a formula like the ones we used in the earlier discussion of different waveforms. You may find it interesting to substitute those formulas in the program and see the waveforms appear on the screen. Since the Amiga's hi-res screen is only 188 pixels high, a vertical difference of one pixel equals an amplitude difference of 2; so the data from a waveform that you draw will be somewhat less precise than if you had computed the waveform mathematically. Nevertheless, the program provides a basis for experimenting with waveforms and can also be fun to use. If you create a waveform that sounds good, use it in your own programs by loading the data from disk into an array or converting the values into DATA statements.

A random waveform produces a rushing or hissing noise and is essential for percussive effects like explosions, the sound of footsteps, and so on. It's difficult, if not impossible, to create "white noise" using SOUND and WAVE commands in Amiga BASIC. Run Program 6-3 and use the mouse to draw random dots all over the waveform display area. No matter how randomly you place the dots, the result is usually a clear, steady tone of one sort or another. This isn't a defect in the computer, but simply a limitation of Amiga BASIC. SOUND and WAVE are easy to use, but don't give you full control over the Amiga's sound-generating machinery. As we'll see in the next section, machine language programmers have access to a much wider range of effects.

Advanced Sound Synthesis

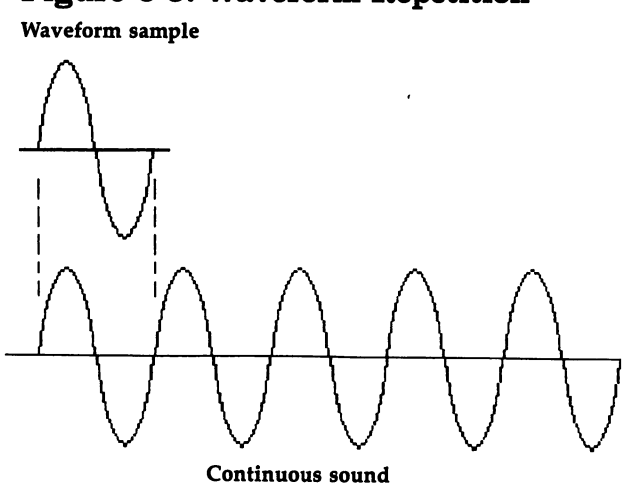
Once you have become familiar with Amiga BASIC sound techniques, you may want to explore more sophisticated, direct methods of sound generation. The Amiga, more than any other home computer, gives you complete control over sounds and can be the equivalent of a miniature electronic sound studio. Unlike some other computers, which produce sounds with simple electronic tone generators (thus providing only a limited number of available waveforms), the Amiga leaves almost every decision up to you. While it may take some time to master this system, the results can be spectacular.

In the remainder of this chapter we'll look more closely at how the Amiga generates sounds and examine a simple machine language sound program. It's worth noting that the type of activity discussed in this section—direct manipulation of hardware registers—is normally considered taboo on a multitasking system like the Amiga. At a higher programming level, the Amiga provides a number of system routines (20, in fact) for opening and closing the audio device, allocating or freeing channels, queuing multiple audio commands, and so forth. The system routines permit you to write sound applications that coexist with other tasks on a friendly basis and don't monopolize system resources unnecessarily. For the most part, however, these routines assume that you already know how to create sound at the hardware level. So this is as good a place to start as any. Chapters 7 and 8 contain more information about using system routines.

Hardware overview. Recall that the Amiga has four separate sound channels, which are mixed down to two stereo outputs. More precisely, each of the four channels is connected to a digital-to-analog (D-to-A) converter circuit which translates digital information (ones and zeros in computer memory) into the analog voltage levels required by conventional sound equipment. The system also includes a low-pass "anti-aliasing" filter which reduces distortion in cases where the sampling frequency (see below) is close to the output frequency for a given tone. Everything else is implemented by software of one sort or another. This provides enormous flexibility, but also complicates the programmer's task.

Waveform sampling. The basic means for sound production on the Amiga is *waveform sampling*. In simple terms, you provide a certain number of bytes of data which represent a waveform and tell the computer how frequently to sample that data set. At each sampling interval, the Amiga extracts the next byte value from the sample and sends it as output to the D-to-A converter for the designated channel; when it reaches the end of the sample, it starts over at the beginning. By repeating this process over and over, a continuous tone is produced. The sampling rate determines the frequency of the tone, and the shape of the waveform data gives it a unique character. The final result is a continuous stream of fluctuating values, which can be pictured as a seamless repetition of the waveform sample's shape (Figure 6-9).

Figure 6-9. Waveform Repetition



As shown earlier, waveform data can take a variety of forms, from the very simple sine wave to complex waves that produce multiple tones through a single channel. From Amiga BASIC, the waveform sample is created by making an integer array and assigned to a given channel with WAVE. When you execute a SOUND command for that channel, the Amiga automatically finds the waveform sample and begins sound production.

Creating sound at the machine level involves the same process, but you are responsible for more of the details. In addition to storing a waveform sample somewhere in memory, you must tell the computer the length of the sample and where it is located. Once this is done, you must tell the computer how frequently to sample the waveform data, set the volume, and then signal that the sound should begin. At this point the sound starts immediately at the specified volume and continues until you take some action to stop it. (This is very different from sound production on certain other computers; the Commodore 64, for instance, generates a characteristic amplitude envelope that causes the tone to fade out after a certain interval.) On the Amiga, when you want the sound to stop, you must tell the computer to cut if off. It might seem like a lot of work, but each step is actually quite easy. Here is the basic procedure:

1. Create waveform sample.
2. Define location of sample.
3. Define length of sample.
4. Set volume.
5. Set sampling rate.
6. Start sound production.
7. Stop sound production.

Note that steps 1–4 are preparatory: They must be performed before making the first sound, but need not be repeated for subsequent sounds unless you want to change the volume, switch to a different waveform, and so forth. Since the sampling rate (step 5) sets the pitch of the sound, it will often be changed each time a new sound is produced. Steps 2–7 are performed by storing values in the Amiga's sound control registers, which are illustrated in Table 6-7.

Table 6-7. Sound Control Registers

Register	Address	Function
BASE	= $DFF000	Chip base address
DMACONW	= BASE + $96	DMA write control
ADKCONW	= BASE + $9E	Audio modulation write control
AUD0LCH	= BASE + $A0	Channel 0 waveform address (high 3 bits)
AUD0LCL	= BASE + $A2	Channel 0 waveform address (low 15 bits)
AUD0LEN	= BASE + $A4	Channel 0 length of waveform sample
AUD0PER	= BASE + $A6	Channel 0 sampling period (124–65535)
AUD0VOL	= BASE + $A8	Channel 0 volume (0–64)
AUD0DAT	= BASE + $AA	Channel 0 output data buffer
AUD1LCH	= BASE + $B0	Same as above for channel 1
AUD1LCL	= BASE + $B2	
AUD1LEN	= BASE + $B4	
AUD1PER	= BASE + $B6	
AUD1VOL	= BASE + $B8	
AUD1DAT	= BASE + $BA	
AUD2LCH	= BASE + $C0	Same as above for channel 2
AUD2LCL	= BASE + $C2	
AUD2LEN	= BASE + $C4	
AUD2PER	= BASE + $C6	
AUD2VOL	= BASE + $C8	
AUD2DAT	= BASE + $CA	
AUD3LCH	= BASE + $D0	Same as above for channel 3
AUD3LCL	= BASE + $D2	
AUD3LEN	= BASE + $D4	
AUD3PER	= BASE + $D6	
AUD3VOL	= BASE + $D8	
AUD3DAT	= BASE + $DA	

In Table 6-7, the register addresses are expressed as an offset from the base address $DFF000. Thus, the actual address of DMACONW is $DFF096; the address of AUD3VOL is $DFF0D8, and so on. These are all fixed locations in the Amiga's memory, which you can access with MOVE or similar machine language instructions. The DMACONW register—the main control register—is used each time you turn a sound on

or off. ADKCONW is used only for modulating one channel's output with the output from a second channel (see below).

We'll look at DMACONW and ADKCONW more closely when we discuss the example program below. For now, notice that the remaining registers are divided into four groups. Each group of six registers controls one of the four sound channels. The two lower registers in each group (AUD0LCH + AUD0LCL, etc.) work as a pair and can contain an 18-bit number: This is where you store the address of the waveform sample. The next higher register in the group (AUD0LEN, AUD1LEN, etc.) stores the length of the waveform data set. The value you put here must be the length of the sample in 16-bit *words*, not 8-bit bytes. For a 20-byte waveform sample, you would store the value 10, and so on. The next higher register (AUD0PER, AUD1PER, etc.) sets the sampling period; this controls the speed at which the waveform is output and thus determines the sound's frequency. After that comes the volume register (AUD0VOL, AUD1VOL, etc.), which can accept values in the range 0–64, with 64 being the maximum volume. The highest register in each channel group (AUD0DAT, AUD1DAT, etc.) is the data buffer for that channel. In ordinary circumstances you won't need to worry about this register; it's used by the Amiga to output data automatically while a sound is in progress. However, if you want to experiment with non-DMA sound production, this register provides a pipeline to the D-to-A converter for an individual channel.

What is audio DMA? The DMA in names like DMACONW stands for *Direct Memory Access*, the process by which data from a waveform sample is continuously output to a sound channel. The nitty-gritty details of DMA are of interest only to advanced programmers. For ordinary sound production, you don't need to know exactly how the system times the interrupts and does all the other jobs needed to make this DMA happen. If you put the right values in the sound control registers, the Amiga will handle the rest. But since DMA appears frequently in Amiga literature, you should at least recognize the term; in this context, a phrase like *enable DMA* means "enable sound production by direct memory access," and so on. From a programming viewpoint, DMA means that once a sound begins, the processor is largely free for other tasks—just as in Amiga BASIC, the sound continues in the background while other program events occur.

Machine Language Sound Program

Program 6-4 illustrates how to generate sounds in machine language, using two audio channels and a variety of frequencies. You'll need a 68000 machine language assembler to enter Program 6-4. If you don't have an assembler, but still want to use the program, you can create the machine code in Amiga BASIC with Program 6-5.

Program 6-5 creates a machine language program on disk by READing DATA values and writing them to a disk file.

Like other machine language programs, Program 6-4 is run by typing its name at the AmigaDOS CLI prompt. Type the program's filename, then enter several letters or other characters (any characters will do, including lowercase, uppercase, and punctuation), and press RETURN: The Amiga plays two notes for each character you supply, changing the frequencies for different characters. The first note is played in channel 0; the second is an octave higher than the first and issues from channel 1. Though its intrinsic value is slight, the program does illustrate the basic mechanics of sound production and can serve as a template for more elaborate experiments. Let's take a closer look at how it works.

The source code begins by defining the addresses and constants we'll need later on. Only two channels (0 and 1) are used. The waveform sample, 32 bytes in length, is defined at the end of the code. Just as in Amiga BASIC, this sample must be a series of bytes in the range −127 to +128. However, the length of the sample is up to you. For beginning experiments, you'll probably find it easier to work with short blocks of data. To minimize "pop" sounds when a sound starts and stops, it's desirable to define the waveform sample so that it begins and ends at zero amplitude.

The program starts (at *Main:*) by computing the ending address of the command line string (the characters you typed after the filename from the CLI prompt) and storing that information in register A1 for later use. Whenever a program is called from the CLI with additional arguments, register A0 contains the address of the argument string, and D0 contains its length. Adding D0 to A0 tells you where the argument string ends. This makes it very easy to pass information from the AmigaDOS CLI environment to a machine language program.

At *Setup:* we handle preparatory jobs that need to be done only once during this execution of the program. The LEA (Locate Effective Address) instruction tells us where the waveform data is located. Since this code, like most 68000 programs, is relocatable, LEA lets us determine where the waveform data ends up, after the computer loads the program from disk. Both channels will use the same waveform, so we store the resulting address in both location registers at once. Note that the destination is the lower register of the location pair (AUD0LCH, not AUD0LCL, etc.). To use two different waveforms, you would need to provide a second waveform sample and perform this process separately for each channel. Next, we store the length of the waveform data set in the AUD0LEN and AUD1LEN registers. Remember, the length equals the number of *words* in the sample, not the number of bytes (even though the data itself consists of byte values). The last preliminary is to set the volume for both channels.

At *Processline:* the program retrieves one character from the command line argument, checks to see that it's within a certain range, and performs some simple arithmetic to convert its ASCII value into a period number that will produce an audible note. If the character is out of range (a space, carriage return, and so forth), we skip the sound-generating code and proceed to the next character, until the entire argument has been processed (see *Nonote:*). Though the results in this case are fairly trivial, you can employ the same general string-processing technique in any program that accepts arguments from the CLI.

Once we have legal period values, we store them in the period registers AUD0PER and AUD1PER to set the sampling period for the current note. When you enable audio DMA for a channel, the Amiga uses the period number as a counter to determine how many clock ticks (0.279365 microseconds) must elapse during each sampling interval. Each time the counter counts down to zero, the computer retrieves another byte value from the waveform sample and sends it as output to the D-to-A converter via that channel's data register (AUD0DAT, and so on). When you select a small period number, the counter counts down rapidly, and the waveform is repeated frequently, generating a high frequency sound. Larger period numbers cause the counter to count down more slowly: Since the waveform is repeated less often, a lower frequency

sound results. Because audio samples are retrieved as a background process (during video scan intervals), there's a limit to how rapidly the computer can retrieve data. The fastest practical retrieval rate is 28,867 samples per second, which means you should avoid using any period smaller than 124.

Sound output is enabled by writing to the DMACONW register. In the example program, this is done separately for each of the two channels. Each of the bits in DMACONW has a different purpose; Table 6-8 explains the bits that control sound production.

Table 6-8. DMACONW Register ($DFF096)

Bit	Label	Purpose
15	SETCLR	Set 1 bits
9	DMAEN	Must be set to 1 for any DMA to take place
3	AUD3EN	Enable/disable channel 3
2	AUD2EN	Enable/disable channel 2
1	AUD1EN	Enable/disable channel 1
0	AUD0EN	Enable/disable channel 0

The four lowest bits of DMACONW determine which channel is affected: To start audio DMA in channel 0, you would write a 1 in bit 0. To stop the sound in channel 0, write a 0 in bit 0, and so on. Bit 9 (DMAEN) must be set to 1 in order for sound DMA to occur, and you must also write a 1 to SETCLR at the same time. In Program 6-4, the channels are enabled at *Channel0:* and *Channel1:*. Once the sound has begun, it continues indefinitely. To disable audio DMA in a given channel, write a 0 to its enable bit in DMACONW.

The remainder of the program is quite straightforward. The *Timeout:* routine delays execution with a do-nothing loop, long enough for the sound to become audible. The simple method used here has a significant drawback: While it's busy timing the delay, the processor can't do anything else. If you want to perform other program tasks while a sound is in progress, you'll need to time the delay with an interrupt or some other scheme.

Modulation. The Amiga can also use the output from one audio channel to *modulate* the sound from another channel. A modulating channel makes no sound of its own; instead, it af-

fects the sound coming out of the other channel. If you recall how a sine wave looks, you can imagine modulation as one channel having control of the "volume knob" of another channel. Assume that channel 1 is set to modulate channel 2. If channel 1 uses a sine wave, the loudness of channel 2 will be turned up and down following the pattern of the sine wave. This creates an effect called tremolo and mimics the wavering sound achieved, for example, when a violinist causes his fingers to tremble against the strings.

There are, in fact, two types of modulation. *Amplitude* modulation (AM), which we've just discussed, affects the volume of the modulated channel, which is useful for envelope generation and tremolo effects. *Frequency* modulation (FM) affects the period of the modulated channel, which changes the output frequency for vibrato and other frequency-based effects.

Your choices for modulation are limited: Any given channel can modulate only its next higher neighbor. Channel 0 can modulate channel 1, but not any other channel. Channel 1 can modulate only channel 2, and channel 2 can modulate only channel 3. Since channel 3 has no higher neighbor, it can't modulate anything; similarly, because channel 0 has no lower neighbor, it cannot be modulated by any other channel. The ADKCONW register controls modulation (Table 6-9).

Table 6-9. ADKCONW Register ($DFF09E)

Bit	Label	Purpose
0	ATVOL0	Channel 0 modulates volume of channel 1
1	ATVOL1	Channel 1 modulates volume of channel 2
2	ATVOL2	Channel 2 modulates volume of channel 3
3	ATVOL3	Channel 3 modulates nothing (output disabled)
4	ATPER0	Channel 0 modulates period of channel 1
5	ATPER1	Channel 1 modulates period of channel 2
6	ATPER2	Channel 2 modulates period of channel 3
7	ATPER3	Channel 3 modulates nothing (output disabled)
15	SETCLR	Set 1 bits

You can use the two types of channel modulation either singly or in combination. If you write a 1 into bit 0 of ADKCONW, channel 0's output modulates the volume of channel 1. If you write a 1 into bit 1, channel 0's output

modulates channel 1's period (thus affecting its frequency). If both bits 0 and 4 are set, channel 0's output modulates channel 1's volume *and* period simultaneously. Just as with DMACONW, you must write a 1 to SETCLR whenever you want to set any other bit in the register to 1. Because ADKCONW is a multipurpose control register (it also plays a role in disk access), you should be careful not to disturb any of its other bits.

When you cause one channel to modulate another, its own output is diverted from sound production to this other task—so it no longer produces sound of its own. Bits 3 (ATVOL3) and 7 (ATPER3) seem to have been included chiefly for the sake of symmetry: Since channel 3 can't modulate anything, the only effect of setting these bits is to disable channel 3's output, which might just as well be done in the usual way.

Using a channel as a modulator is very similar to using it as a sound generator except that its output is rerouted and used for a different purpose. Just as in the normal case, you must set its period to determine how frequently to sample the data, enable DMA to begin output, and so on. You must also create a data set and tell the computer where to find it. But the Amiga interprets the contents of the modulator's data sample quite differently. As noted above, for sound production the waveform sample data is treated as a series of 8-bit bytes. When you use a channel as a *volume* modulator, its data set is treated as a series of 16-bit words, but only the low 7 bits of each data word are significant:

Bits	**Purpose**
0–6	Volume data
7–15	Ignored

When you use a channel as a *period* modulator, its data sample is interpreted as a series of 16-bit words, every bit of which is significant:

Bits	**Purpose**
0–15	Period data

When a channel is used to modulate *both* volume and period, the interpretation flip-flops between alternate words: The first word is read as seven bits of volume information, and the second is interpreted as a word of period data; words 3, 5, 7,..., are treated as volume data, words 4, 6, 8,..., are treated as

period data, and so on. Don't confuse the modulator's data set with the waveform data sample for the modulated channel: The two are independent and each can be whatever length is needed.

While modulation is handy for many purposes, one of the most obvious uses for it is envelope generation. Every sound has a characteristic envelope or amplitude pattern which determines how quickly it rises to its maximum volume, how long it exists, and how quickly it fades back into silence. Some computers (the Commodore 64, for instance) generate sound envelopes automatically, but the Amiga does not. In theory, envelope generation is a simple task: All you need to do is change a channel's volume according to a specified pattern while the channel is producing a sound. In practice, monitoring the progress of individual sounds and deciding when it's time to make another volume change can absorb a lot of processor time. By using a second channel to modulate the amplitude of the first, you can make the computer do most of the work for you. You might find it an interesting project to modify the example program so that channel 0 modulates the volume of channel 1 (creating an amplitude envelope) or channel 1's period (creating a vibrato), rather than producing an independent sound of its own.

Program 6-4. Machine Language Sound

```
* CLI_Sound.asm  *  Machine language sound example
* Accept CLI arguments, generate tones in 2 channels

Chipadr equ $dff000        ;base address of chip

DMAconw equ Chipadr+$96    ;DMA control write register

Aud0Lc  equ Chipadr+$a0    ;Channel 0 waveform sample location
Aud0Len equ Chipadr+$a4    ;Channel 0 waveform sample length
Aud0per equ Chipadr+$a6    ;Channel 0 period
Aud0vol equ Chipadr+$a8    ;Channel 0 volume

Aud1Lc  equ Chipadr+$b0    ;Channel 1 waveform sample location
Aud1Len equ Chipadr+$b4    ;Channel 1 waveform sample length
Aud1per equ Chipadr+$b6    ;Channel 1 period
Aud1vol equ Chipadr+$b8    ;Channel 1 volume

Setclr  equ $008000        ;DMA SETCLR bit
OffDMA  equ 0              ;Disable DMA
Aud0en  equ $01            ;Enable/disable Channel 0
Aud1en  equ $02            ;Enable/disable Channel 1
DMAen   equ $0200          ;Enable DMA
Count   equ $ffff          ;Counter for delay timer

Main:
        add.l a0,d0          ;Find and record the ending
        move.l d0,a1         ;address of command tail.

Setup:
        lea Wavesample,a2  ;Find address of waveform sample data.
```

```
        move.l a2,Aud0Lc    ;Set waveform location Channel 0.
        move.l a2,Aud1Lc    ;Set waveform location Channel 1.

        ;Length = number of WORDS (not bytes) in waveform sample.
        move.w #16,Aud0Len ;Set sample length Channel 0.
        move.w #16,Aud1Len ;Set sample length Channel 1.

        ;Volume can range from 0-64
        move.w #64,Aud0vol ;Set Channel 0 volume at maximum.
        move.w #64,Aud1vol ;Set Channel 1 volume at maximum.

Processline:
        move.w #0,d0
        move.b (a0),d0      ;Get new character, and
        cmpi.b #'!',d0      ;check if within desired range.
        blt Nonote          ;Reject ASCII values under '!'
        cmpi.b #'}',d0      ;Check top of range
        bgt Nonote          ;Reject if above '}'
                            ;Derive period value based on
                            ;ASCII value of current character.
        mulu #4,d0          ;Period can range from 124-65535.
        move.w d0,Aud0per   ;Set channel 0 period.
        mulu #2,d0
        move.w d0,Aud1per   ;Set channel 1 period.

        ;Everything's ready--let's make some sounds...

ChannelL0:
        move.w #(Setclr+DMAen+Aud0en),DMAconw  ;Enable channel 0.
        bsr.Timeout                            ;Delay.
        move.w #(OffDMA+Aud0en),DMAconw        ;Disable channel 0.
```

```
ChannelL1:
          move.w #(Setclr+DMAen+Audlen),DMAconw  ;Enable channel 1.
          bsr Timeout                            ;Delay.
          move.w #(OffDMA+Audlen),DMAconw        ;Disable channel 1.
Nonote:
          addq.l #1,a0  ;Process every character of argument string
          cmpa.l a0,a1
          bne Processline
          rts

Timeout:
          move.w #2,d0
          move.l d1,-(a7)
          subq.w #1,d0
Moon:
          move.w #Count,d1
Loon:
          dbra d1,Loon
          dbra d0,Moon
          move.l (a7)+,d1
          rts

Wavesample:
          dc.b 0,32,64,96,120,120,120,120,120,120,120,120
          dc.b 120,120,120,120,-120,-120,-120,-120,-120,-120
          dc.b -120,-120,-120,-120,-120,-120,-96,-64,-32,0

          END
```

Program 6-5. Machine Language Filemaker

```
'Filemaker for machine language sound example

FOR j=Ø TO 255
  READ Value
  MLcode$=MLcode$+CHR$(Value)
  Checksum=Checksum+Value
NEXT

IF Checksum<>22825 THEN
  PRINT "Error in data statements."
  END
END IF

INPUT "Filename for disk file";filename$
OPEN filename$ FOR OUTPUT AS #1
PRINT #1, MLcode$
CLOSE 1

PRINT "To run ML program, enter SYSTEM to exit to D
OS."
PRINT "Then open a CLI window and enter the filenam
e"
PRINT "you used when creating the file, followed by
"
PRINT "a space and any other characters."

DATA  Ø , Ø , 3 , 243 , Ø , Ø , Ø , Ø
DATA  Ø , Ø , Ø , 1 , Ø , Ø , Ø , Ø
DATA  Ø , Ø , Ø , Ø , Ø , Ø , Ø , 5Ø
DATA  Ø , Ø , 3 , 233 , Ø , Ø , Ø , 5Ø
DATA  2Ø8 , 136 , 34 , 64 , 69 , 249 , Ø , Ø
DATA  Ø , 168 , 35 , 2Ø2 , Ø , 223 , 24Ø , 16Ø
DATA  35 , 2Ø2 , Ø , 223 , 24Ø , 176 , 51 , 252
DATA  Ø , 16 , Ø , 223 , 24Ø , 164 , 51 , 252
DATA  Ø , 16 , Ø , 223 , 24Ø , 18Ø , 51 , 252
DATA  Ø , 64 , Ø , 223 , 24Ø , 168 , 51 , 252
DATA  Ø , 64 , Ø , 223 , 24Ø , 184 , 48 , 6Ø
DATA  Ø , Ø , 16 , 16 , 12 , Ø , Ø , 33
DATA  1Ø9 , Ø , Ø , 7Ø , 12 , Ø , Ø , 125
DATA  11Ø , Ø , Ø , 62 , 192 , 252 , Ø , 4
DATA  51 , 192 , Ø , 223 , 24Ø , 166 , 192 , 252
DATA  Ø , 2 , 51 , 192 , Ø , 223 , 24Ø , 182
DATA  51 , 252 , 13Ø , 1 , Ø , 223 , 24Ø , 15Ø
DATA  97 , Ø , Ø , 38 , 51 , 252 , Ø , 1
DATA  Ø , 223 , 24Ø , 15Ø , 51 , 252 , 13Ø , 2
DATA  Ø , 223 , 24Ø , 15Ø , 97 , Ø , Ø , 18
DATA  51 , 252 , Ø , 2 , Ø , 223 , 24Ø , 15Ø
DATA  82 , 136 , 179 , 2ØØ , 1Ø2 , 168 , 78 , 117
```

```
DATA  48 , 60 , 0 , 2 , 47 , 1 , 83 , 64
DATA  50 , 60 , 255 , 255 , 81 , 201 , 255 , 254
DATA  81 , 200 , 255 , 246 , 34 , 31 , 78 , 117
DATA  0 , 32 , 64 , 96 , 120 , 120 , 120 , 120
DATA  120 , 120 , 120 , 120 , 120 , 120 , 120 , 120
DATA  136 , 136 , 136 , 136 , 136 , 136 , 136 , 136
DATA  136 , 136 , 136 , 136 , 160 , 192 , 224 , 0
DATA  0 , 0 , 3 , 236 , 0 , 0 , 0 , 1
DATA  0 , 0 , 0 , 0 , 0 , 0 , 0 , 6
DATA  0 , 0 , 0 , 0 , 0 , 0 , 3 , 242
```

Chapter 7

C Programming

Marc B. Sugiyama and Christopher D. Metcalf

C Programming

Marc B. Sugiyama and Christopher D. Metcalf

The C programming language was developed in the early 1970s by Dennis Ritchie, then an employee of Bell Laboratories. C evolved out of a language called B, which was itself developed from a language called BCPL. It is designed to be a programmer's language, not a learner's language like BASIC or Pascal. It is powerful and versatile enough to implement everything from low-level operating systems functions to utility programs to applications. Unix, MS-DOS, and Digital's VMS operating systems were all developed in C or C-like languages.

What Is C?

C is a *compiled* language. When you program in C, you run your program through a *compiler.* The compiler converts the program into a machine language equivalent which the computer can execute directly. Other languages, such as BASIC and LISP, are *interpreted;* the computer executes the original code on the fly. Compiled languages tend to be fast; interpreted languages tend to be very easy to edit and debug. Compiled programs are still slower than their assembly language equivalents; however, they are more easily written and debugged than assembly language programs.

C is both a high-level and a low-level language. It has many features you might expect in high-level languages, such as strings, file handling, and floating-point math. It also has many other instructions, such as bit operations (AND, OR, bit shifting), which are more likely to be found in low-level assembly languages. However, unlike an assembly language, which is specific to one processor, C programs are easily ported between many different computers. C compilers can be found on computers ranging from large VAX and IBM mainframes to Commodore 64s.

C code is cryptic at best. In a sense this is good, as it allows very short programs to be very powerful. However, it is just as easy to produce very structured, organized code as it is to produce code which is completely unstructured and chaotic. C can be all things to all people.

These features of C are some of the reasons behind its immense popularity. C's popularity is also due in part to the increased availability of good C compilers for personal computers and the increased accessibility of large minicomputers.

C's Availability on the Amiga

The C compiler is available on the Amiga from two sources. Commodore provides a native C compiler and a cross compiler from the IBM PC to the Amiga in its developer's package. The native compiler runs on the Amiga and produces code which the Amiga can execute. The cross compiler, on the other hand, runs on the IBM-PC, and the code it produces must be transferred to the Amiga before it can be run and tested. Both of the Commodore-supplied compilers were developed by Lattice. The other C compiler is called AZTEC-C and is available from Manx. They, too, offer both native and cross compilers.

The Lattice C Compiler

The Lattice C compiler is actually two separate programs: LC1 and LC2. LC1 is run on C source code to produce an intermediate *quad* file. LC2 is then run on this file in order to generate an *object module*, which contains the executable code represented by your C program. However, your program will no doubt contain references to built-in C procedures (the *printf* routine, for example). A *linker* program, called ALink on the Amiga, is needed to link your object module with the *library* modules containing the C standard functions (LC.LIB) and the modules containing Amiga-specific functions (AMIGA.LIB). So, the compilation process is really three-step.

To simplify this, a simple CLI script file called *make* is provided that calls the compiler and linker programs with the appropriate parameters. All of the example programs presented here were written using the Lattice C compiler, version 3.02, as supplied with the Amiga developer's kit. For more information about the various compiler options see Appendix D.

Compiling a Lattice C Program

This section will explain the actual compilation process. On the way, we'll show you how to set up a programming environment similar to the one we are most comfortable using. You are certainly free to build your own environment if you don't approve of ours.

Our environment is based on the ability of AmigaDOS to run script files—files which hold commands to be executed as if they were being typed from the keyboard (like VMS .COM files or PC-DOS .BAT files). When you boot the Amiga, AmigaDOS runs a script file called *Startup-Sequence* in the *s* directory of the boot disk. Our startup-sequence includes the following commands:

```
assign i: csys:include
assign lib: csys:lib
assign lc: csys:c
stack 8000
makedir ram:t
date ?
```

We've named the compiler disk CSYS:. It has an *include* directory for the include files and a *lib* directory for the libraries. The Startup-Sequence performs the following operations: It assigns the logical device i: to the include directory on csys:, so we can access csys:include with just i:. In a similar way, it assigns the logical devices lib: and lc: to particular directories of csys:. Lattice recommends a default stack larger than 4096, so we use 8000. Finally, we build a directory for temporary files on the RAM disk and ask for the date.

When you're working on a program, we recommend putting source code in the RAM disk while you're compiling. Just copy the program to the ram: device. This decreases the compilation time dramatically.

Now that we have everything set up, we're up to compiling the program. There are several ways to invoke the Lattice C compiler. We've built two script files which aid compilation. The first is called "cc" (Program 7-2). It will compile a program from start to finish by invoking both passes of the compiler and the linker. If you like, you can use cc on the sample program "Mandelbrot.c" at the end of this chapter.

But cc will work only if you are compiling a program which doesn't have to be linked with other object modules. The "hello" programs must be linked with the object module "xopenscreen" to work properly. Before you can produce executable copies of the hello programs, you must first build the xopenscreen object module. This is simpler than it sounds. All you have to do is run the compiler without linking. The .o file produced by the second pass compiler is the object module. The source code for xopenscreen is Program 7-1.

We've written a script file called "ccnl" (cc with no linking), Program 7-3. You have to use it on both the hello source you're compiling and the xopenscreen source. Now that you've made two object modules, you have to run alink to put them together into an executable file. You can use cc as a template for the correct format of the alink command.

Below is a sample session in which we compile the program hello.c. The boldface text is the text which is to be typed by you. Note that both compilations produce a number of warnings. These can be ignored for this program. You can eliminate these warnings by including additional header files.

```
1>cd ram:
1>copy work:xopenscreen.c ram:
1>execute ccnl xopenscreen

Lattice AMIGA 68000 C Compiler (Phase 1)
V3.02 Copyright (C) 1984 Lattice, Inc.
xopenscreen.c 39 Warning 85: function return value mismatch
xopenscreen.c 40 Warning 61: undefined structure tag "Region"
xopenscreen.c 40 Warning 61: undefined structure tag "CopList"
xopenscreen.c 40 Warning 61: undefined structure tag "UCopList"
xopenscreen.c 40 Warning 61: undefined structure tag "cprlist"
xopenscreen.c 40 Warning 61: undefined structure tag "VSprite"
xopenscreen.c 40 Warning 61: undefined structure tag "collTable"
xopenscreen.c 40 Warning 61: undefined structure tag "Device"
xopenscreen.c 40 Warning 61: undefined structure tag "Unit"
xopenscreen.c 40 Warning 61: undefined structure tag "KeyMap"

Lattice AMIGA 68000 C Compiler (Phase 2)
V3.02 Copyright (C) 1984 Lattice, Inc.
Module size P=00000086 D=00000000 U=00000000
1>copy work:hello.c ram:
1>execute ccnl hello

Lattice AMIGA 68000 C Compiler (Phase 1)
V3.02 Copyright (C) 1984 Lattice, Inc.
i:exec/types.h 52 Warning 84: redefinition of pre-processor symbol "NULL"
hello.c 65 Warning 61: undefined structure tag "Region"
hello.c 65 Warning 61: undefined structure tag "CopList"
hello.c 65 Warning 61: undefined structure tag "UCopList"
hello.c 65 Warning 61: undefined structure tag "cprlist"
hello.c 65 Warning 61: undefined structure tag "VSprite"
hello.c 65 Warning 61: undefined structure tag "collTable"
hello.c 65 Warning 61: undefined structure tag "Device"
hello.c 65 Warning 61: undefined structure tag "Unit"
hello.c 65 Warning 61: undefined structure tag "KeyMap"
hello.c 65 Warning 61: undefined structure tag "GfxBase"
hello.c 65 Warning 61: undefined structure tag "IntuitionBase"
```

```
Lattice AMIGA 68000 C Compiler (Phase 2)
V3.02 Copyright (C) 1984 Lattice, Inc.
Module size P=000001A0 D=0000010D U=00000010
1>alink lib:lstartup.obj+hello.o+xopenscreen.o library lib:lc.lib+lib:am
   iga.lib to hello map nil:

Amiga Linker Version 3.18.
Copyright (C) 1985 by Tenchstar Ltd., T/A Metacomco.
All rights reserved.
Linking complete - maximum code size = 11632 ($00002D70) bytes
```

In the newer version of the Lattice compiler (version 3.03), the lstartup.obj file has been renamed c.o. Furthermore, adding the –v flag to lc2 improves the performance of the c program by about 20 percent.

To reduce typing, we often give ram: source files short names. For example, xopenscreen.c might have been called x.c during development. Note that once you've made the object file of xopenscreen.c, there's no need to recompile it (unless, of course, you've changed the source code) to use with other programs. From now on, you just use the object module xopenscreen.o.

Program 7-1. xopenscreen.c

```
#include <exec/types.h>
#include <intuition/intuition.h>
/*
 * XOpenScreen()
 *
 * This function is intended to be compiled and linked to other
 * functions which need to open simple screens.  The intent is to simplify
 * the OpenScreen() procedure.
 *
 * Assumptions:
 *
 *screen occupies full width and height of display
 *BlockPen = 1, DetailPen = 0
 *CUSTOMBITMAP mode is disallowed
 *the default "Topaz" font is used
 *no custom gadgets are allowed
 *
 */

struct Screen *XOpenScreen(Title, ViewModes, Depth)
UBYTE *Title; /* pointer to the title (a string); or NULL */
UWORD ViewModes;/* HIRES, LACE, SPRITES, DUALPF, HAM */
WORD Depth;/* depth of screen (1-6) */
{
struct NewScreen XNewScreen;
XNewScreen.LeftEdge = 0;
XNewScreen.TopEdge = 0;
XNewScreen.Width = ViewModes & HIRES ? 640 : 320;
XNewScreen.Height = ViewModes & LACE ? 400 : 200;
XNewScreen.Depth = Depth;
```

```
XNewScreen.DetailPen = Ø;
XNewScreen.BlockPen = 1;
XNewScreen.ViewModes = ViewModes;
XNewScreen.Type = CUSTOMSCREEN;
XNewScreen.Font = NULL;
XNewScreen.DefaultTitle = Title;
XNewScreen.Gadgets = NULL;
XNewScreen.CustomBitMap = NULL;
return (OpenScreen(&XNewScreen));/* open screen */
}
```

Program 7-2. CC

```
.Key file,opt1,opt2,opt3 lc1 <opt1> <opt2> <opt3> -ii: -ii:lattice/ <file$test>
if not exists "<file$test>.q"
echo "Compile failed."
quit 20
endif
lc2 <file$test>
alink from lib:lstartup.obj+<file$test>.o lib lib:lc.lib,lib:amiga.lib to <file$test> map nil:
```

Program 7-3. CCNL

```
.Key file,opt1,opt2,opt3 lc1 <opt1> <opt2> <opt3> -ii: -ii:lattice/ <file$t1>
if not exists "<file$t1>.q"
echo "Compile failed."
quit 20
endif
lc2 <file$t1>
```

C Programming on the Amiga

From this point on, we're going to assume that you're familiar with the C programming language. As with any computer language, it would take an entire book to teach you C. Thus, we direct you to one of the multitude of introductory-level C books from which you can learn this language. Please refer to the end of the chapter for a list of references. What follows is an overview of C programming on the Amiga.

C programming on the Amiga is very straightforward. If you glance at the sample programs, you might at first be overwhelmed by the large and complex data structures. The routines which deal with those huge data structures are surely more complex than the data structures themselves, right? Wrong. Most Amiga programming simply requires building data structures and passing them to one of the Amiga's system routines. Nothing is really all that hard to do. What's hard is deciding what you *want* to do, not telling the Amiga *how* to do it.

Header files. All the data structures that the Intuition library operates on are defined in the file intuition/intuition.h. This file, which is a header file, must be included in your program with the #include statement. You must also include exec/types.h since the intuition/intuition.h file uses symbols defined in this file. See Programs 7-3, 7-4, 7-5, and 7-6 for examples.

Libraries

Many of the Amiga's built-in functions are designed to handle its hardware and data structures. These routines are maintained in objects referred to as *libraries.* A library is simply a set of routines which are tied together as a logical set. They are designed to be used by many different programs simultaneously. Thus, each task doesn't have to have its own identical set of library functions in memory at the same time. Don't confuse these libraries with the ones which are linked to your C object modules. The libraries we're referring to here are accessed during the execution of the program and are not part of the program itself.

The libraries are either loaded from disk or are already resident in memory (some, like the Intuition library, are even stored in ROM). Once a library is loaded off disk, it is not purged from memory unless the memory is required by the system and no program is still using the library. In other

words, the libraries remain resident in memory as long as possible. This improves system performance by keeping disk activity at a minimum.

To access routines stored in a library, you must call the routine **OpenLibrary( ).** This returns a pointer to a Library structure, which contains a table of addresses where the routines in the library can be found. The **OpenLibrary( )** function will cause the library to be loaded into memory if it hasn't already been loaded. It also tells the library manager that your task requires access to the library. When you are finished with the library, you must close it with the **CloseLibrary( )** function, which informs the library manager that your task no longer requires access to that particular library. If you do not close the library, the librarian will never realize that your task no longer requires access and will not free memory allocated to the library when so requested. Although it's not really necessary to close a ROM-based library (since it's always in memory), it's generally good practice to do so should future versions of the Amiga operating system use all RAM-based libraries.

Graphics

There are two libraries which are commonly used to handle graphics. This essentially divides graphics handling in the Amiga into two different levels. At the top level are the Intuition routines. These routines are contained in *intuition.library* and are used by Intuition and the Workbench to open and maintain the windowing environment. Below Intuition are the routines contained in *graphics.library.* The graphics library is technically part of the Kernel (to make it distinct from Intuition). These routines draw the lines and fill the blocks which constitute the windows. Both the Intuition library and the graphics library are in "ROM"; that is, they are loaded off the Kickstart disk when the system is powered up.

When you open the graphics library and the Intuition library, you must use the names GfxBase and IntuitionBase, respectively, for the library pointers. The compiler library (Amiga.lib) assumes that these are the names being used to reference routines in these libraries. The graphics library and Intuition library are used together to manipulate the four basic units of Amiga display handling: screens, windows, gadgets, and menus.

Screens

Screens act as the surface for all of your work. They are the background on which all of the windows appear. You're allowed to have more than one screen open at a time. They are dragged up and down like blackboards and can be depth-arranged like windows. When you open a new screen, there are two key parameters that you need to select.

The first parameter is the resolution. The Amiga supports two different horizontal resolutions (320 or 640) and two vertical resolutions (200 or 400). Thus, you have four possible screen modes: 320 × 200, 640 × 200, 320 × 400, and 640 × 400. Amiga documentation refers to the 640 horizontal mode as high resolution (HIRES), and the 400 vertical mode as interlaced (LACE). Interlaced screens can flicker on all but the most expensive monitors and thus should be used only when necessary. Note that the lower the resolution, the smaller amount of memory the screen requires and the faster the system will perform.

The second parameter which can be changed is the "depth." This is simply the number of bit planes which should be allocated for the screen. Thus, if the depth of the screen is 3, you're allocating three bit planes, and you are allowed eight colors. Don't confuse this kind of depth with depth as in "depth arrangement." Depth arrangement is simply the stacking order of the windows or screens (which screen appears on top of the others). Note that the greater the number of bit planes, the more memory the system requires and the slower the system will perform, but the more colors you will have available at one time. The default Workbench screen is 640 × 200 by 2 bit planes. This provides four colors and good system performance.

Now that you've decided on the resolution and depth of your custom screen, you have to inform the Amiga operating system. You begin by filling the appropriate data into the NewScreen structure. This structure is required only when you open the screen and can be discarded after the screen has been created.

```
struct NewScreen {
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
    UBYTE DetailPen, BlockPen;
    USHORT ViewModes, Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
```

```
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap; }
```

The fields in this structure have the following meanings:

LeftEdge	Position of the left edge of the screen. The current version of the Amiga hardware doesn't support this feature. It should be set to zero for upward compatibility.
TopEdge	Position of the top edge of the screen. Set this to zero if you want the screen to start at the top of the display.
Width	Width of the screen. This should either be 320 or 640 (remember, 640 is the high-resolution mode).
Height	Height of the screen. This should either be 200 or 400 (remember, 400 is the interlace mode).
Depth	Number of bit planes to use (only the values 1–6 are valid).
DetailPen	Color register to use for details such as the gadgets and the text in the title bar. Generally, this is set to zero.
BlockPen	Color register to use for block fills such as the title bar area. Generally, this is set to one.
ViewModes	These flags are used to select the proper display mode. There are currently five flags which can be set. The important ones to us right now are: HIRES—Selects 640 horizontal resolution; 320 horizontal resolution is used if this flag is not specified. INTERLACE—Selects 400 vertical resolution (interlace mode); 200 vertical resolution is used if this flag is not specified.
Type	For our purposes, the screen type is always CUSTOMSCREEN.
Font	This is a pointer to a TextAttr structure. To use the default system font (generally, Topaz eight- or nine-point font), set this pointer to NULL.
DefaultTitle	Pointer to a string which represents the title of the screen. Setting this pointer to NULL makes a screen with no title.
Gadgets	This is a pointer to the first in a linked list of custom (application) screen gadgets. We won't discuss screen gadgets in this chapter; however, window gadgets are discussed below.
CustomBitMap	This is a pointer to a BitMap structure. We won't talk about custom bitmaps, so set this to NULL.

For example, if you wanted to open a 640 × 200 screen with eight colors, no screen gadgets, and the default font, you would use the following declaration:

```
struct NewScreen *newscreen = {
    0,0,                /* left and top edge */
    640,200,            /* width and height */
    3,                  /* depth */
    0,1,                /* DetailPen and BlockPen */
    HIRES,              /* ViewMode */
    CUSTOMSCREEN,       /* Type */
    NULL,               /* default font */
    "First Screen",     /* screen's title */
    NULL,               /* no screen gadgets */
    NULL                /* no custom bitmap */ };
```

Once you've built a structure like the one above, you pass a pointer to it to the OpenScreen( ) function. Note that our example NewScreen structure will open a screen with the default font (as set by Preferences). This function returns a NULL if the screen could not be opened (for example, if the Amiga is out of memory); otherwise, it returns a pointer to a Screen structure. This pointer is used to identify the screen when it is used by other routines. For example, to open a window in this screen, we have to tell the OpenWindow( ) function where to find the screen.

You can force the screen to open with the Topaz 60- or 80-column font by using the following TextAttr structure and replacing the NULL in the NewScreen Font field with a pointer to textattr (this is all we are going to say about fonts):

```
struct TextAttr textattr = {
    "topaz.font",       /* font name */
    TOPAZ_EIGHTY,       /* height of font */
    FS_NORMAL,          /* printing style */
    FPF_ROMFONT         /* preferences */ };
```

In order to simplify the process of opening a screen, we have written a short C function called XOpenScreen( ) which performs most of the work for you. All you must specify are the depth and the resolution of the screen, and the routine takes care of the rest. Note that it does make certain assumptions:

- The screen occupies the full width and height of display.
- BlockPen = 1.
- DetailPen = 0.

- CUSTOMBITMAP mode is not allowed.
- The default Topaz font is used (60 or 80 columns as set by the Preferences utility).
- No custom screen gadgets are allowed.

XOpenScreen( ) returns a pointer to a Screen structure if the screen was opened successfully or a NULL, just like the OpenScreen( ) function.

Once your program is done with the screen, you must close it using the CloseScreen( ) function. Pass this function the pointer to the Screen structure returned by the OpenScreen( ) or XOpenScreen( ) command. If you don't close the screen before exiting the program, there will be no way of removing the screen from the system and, consequently, releasing the memory allocated to the screen without rebooting the system.

There are other C functions associated with screens, but they are outside the scope of this discussion.

Windows

Windows are where most of the interaction with the computer is performed. These are where output is generated and input is acquired. Intuition has a variety of functions which handle the resizing, depth arrangement, and movement of windows. Generally, these are handled automatically; the program can be informed and confirm such actions, but it doesn't *have* to be told that such action is taking place. In other words, window resizing, depth arrangement, and dragging can be completely transparent to the program using the window for input and output.

The icons which you click on to perform this magic with the windows are called *gadgets*. The "system" gadgets, those that allow you to drag, depth arrange, close, and resize windows, are only a small example of what gadgets can be used for. In the section of this chapter called "Application Gadgets," we'll discuss how you can add your own gadgets to the system gadgets already attached to the window. As with screens, let's begin with the basics.

Windows are somewhat harder to open than screens, only because their initialization data structure is larger. For windows, you'll have to decide where they are going to go on the screen, how large they are when they're first opened, how big or small they can get (if you allow resizing), and so forth.

Once you figure all this out, you pass this information to Intuition with the OpenWindow( ) function in a NewWindow structure.

```
struct NewWindow {
    SHORT LeftEdge, TopEdge, Width, Height;
    UBYTE DetailPen, BlockPen;
    ULONG IDCMPFlags, Flags;
    struct Gadget *FirstGadget;
    struct Image *CheckMark;
    UBYTE *Title;
    struct Screen *Screen;
    struct BitMap *BitMap;
    SHORT MinWidth, MinHeight, MaxWidth, MaxHeight;
    USHORT Type; }
```

where the fields have the following definitions:

LeftEdge, TopEdge	Where the left-top edge of the window should be placed on the screen when the window is opened. The location (0,0) is the top left corner of the screen.
Width, Height	The size of the window in pixels (screen dots) when the window is first opened.
DetailPen, BlockPen	These are analogous to the NewScreen fields of the same name. If you use −1 for either of these, the default value specified when the screen was opened will be used for the window.
IDCMPFlags	IDCMPs (Intuition Direct Communication Message Ports) are discussed below.
Flags	There are several flags which must be set to tell the operating system something about the window. It is easy to organize the flags into logical groups. Some of the flags control which system gadgets are rendered with the window: WINDOWSIZING—If you specify this flag, Intuition will add a resizing gadget to the lower right corner of the window and allow the user to resize the window. You must specify NOCAREREFRESH if you don't want Intuition to send you a message every time the window is resized. WINDOWDEPTH—This flag forces Intuition to add depth arrangers to the title bar of the window and allow the user to depth arrange this window with the others on the screen. WINDOWCLOSE— Specifying this flag makes Intuition provide a window with a close box. When

the user clicks on the close box, Intuition will send your program an IDCMP of class CLOSEWINDOW. Note that you should also set the CLOSEWINDOW flag in the IDCMP flags section (don't worry if this isn't making sense yet. IDCMPs will be discussed more fully later).

WINDOWDRAG—This tells Intuition that the user is allowed to move the window around the screen.

Generally a Workbench window will have all four of these system gadgets; however, the CLI window doesn't have a close box.

The second set of flags relates to how Intuition should handle refreshing the window when it is overlapped by another window.

SIMPLE_REFRESH—Tells Intuition not to save the portion of the window which is covered by another window. All of the windows share the same screen memory. When a window is covered by another window, the memory holding the image of the old window has been altered to hold the image of the new window. If this flag is set, Intuition will not store the contents of the old window, and any image rendered in it will be lost.

SMART_REFRESH—The opposite of SIMPLE_REFRESH. Specifically, this tells Intuition to save the contents of the window if it is covered by another window. Thus, if you overlap a window with another and then uncover the window again, the uncovered portion of the window will be redrawn as it was before.

The last set of flags is more diverse:

ACTIVATE—Tells intuition that the window is to be made the "active" window when it is opened.

NOCAREREFRESH—Used to tell Intuition that you do not want to receive an IDCMP when the window is resized.

FirstGadget — Pointer to the first entry in a linked list of user-defined gadgets (referred to as both application gadgets and custom gadgets in Amiga documentation). Set this field to NULL if there are no custom gadgets.

CheckMark	Pointer to the image of the checkmark used to identify selected items on the menus. Set this field to NULL if you want to use the system default checkmark.
Title	Pointer to a string which represents the title of the window.
Screen	Pointer to the Screen structure belonging to the screen you want to open the window in. This is obtained from the OpenScreen( ) function. If this field is NULL, then the window is opened in the Workbench screen.
BitMap	Pointer to a "super bitmap." We will not be discussing the use of super bitmaps. This is used only if a special flag is set in the Flags field. Generally, however, you'll want to set this to NULL.
MinWidth, MinHeight, MaxWidth, MaxHeight	The minimum and maximum size of the window allowed by Intuition during resizing. If you use zero for any of these values, then the initial value is taken as the value for that field. Thus, if the initial width of the window is 100, and you specify MinWidth to be 0, then MinWidth is taken as 100.
Type	The current version of Intuition supports only two different types of windows: WBENCHSCREEN—Specify this type of window if you want the window to be opened in the Workbench screen. CUSTOMSCREEN—This type of window is opened in the custom screen pointed to by the Screen field. Thus, you must open a CUSTOMSCREEN before you can open a window in it.

Notice that you do not specify a resolution or a depth for a window as you do for a screen. These parameters are screen dependent, not window dependent. Thus, the resolution and depth of the window are dependent on the resolution and depth of the screen onto which it is opened. If you open a CUSTOMSCREEN of resolution 320 × 400, with a depth of 3, then the windows you open in that screen will have a maximum size of 320 × 400 and can display up to eight colors.

Once you've filled in all of the data, you call OpenWindow( ) with a pointer to the NewWindow structure. If the OpenWindow( ) call was successful, it will return a pointer to the Window structure associated with the new window. The NewWindow structure is no longer needed and can

be discarded or used to open another window. If the OpenWindow( ) wasn't successful (suppose you don't have enough memory), then it returns a NULL. Below is an example of a "typical" window declaration. We aren't allowing for custom gadgets (since we haven't discussed them yet), but we do allow for window resizing, dragging, and depth arrangement. We'll also make Intuition open the window in the Workbench screen.

```
struct NewWindow newwindow = {
    20,20,              /* Left edge and top edge */
    100,100,            /* width and height */
    0,1,                /* detail and block pens */
    NULL,               /* IDCMP flags */
    SMARTREFRESH | ACTIVATE | WINDOWSIZING |
      WINDOWDRAG | WINDOWDEPTH,
                        /* flags */
    NULL,               /* no gadgets */
    NULL,               /* use default checkmark for menus */
    "Simple Sample",    /* window's title */
    NULL,               /* open in Workbench screen */
    NULL,               /* no custom bitmap */
    50,50,              /* min width and height */
    200,200,            /* max width and height */
    CUSTOMSCREEN        /* type */}
```

Using the Kernel Routines in the Window

Now that you have an open window, what can you do with it? Most of the figure drawing is performed by the Kernel. The Kernel is one level beneath Intuition. Its routines generally require that you pass a pointer to a RastPort, a ViewPort, or a View. These quantities are stored in the Window structure returned by the OpenWindow( ) routine. The following might be useful:

- To obtain a RastPort, use the following code: RPort = Window–>RPort, where RPort is a pointer to a RastPort structure and Window is a pointer to a valid Window structure (remember, OpenWindow( ) returns a pointer to a Window structure).
- The function ViewPortAddress(Window) will return the ViewPort for the Window.
- ViewAddress( ) will return the address of the Intuition View structure. This is the head of a linked list of the ViewPortAddress( ) maintained for the Kernel routines.

When dealing with the Kernel routines, it's important to keep the following terminology in mind: The *foreground,* or primary drawing pen, FgPen, is the same as APen; the *background,* or secondary drawing pen, BgPen, is the same as BPen. These don't really correspond to the DetailPen and BlockPen used by Intuition. However, if you want some drawing to be done in the same pen as the DetailPen or the BlockPen, then you can set the APen or BPen to the proper color register before performing the draw. The sample programs have numerous examples of how the Kernel routines can be used to manipulate the windows. The "hello" programs (for example) use Kernel routines to position the cursor and write text to the window, and the Mandelbrot program (the final example) uses Kernel routines to draw the Mandelbrot set on the window.

A final note with windows: As with screens, it's very important that a program close all of the windows it has opened. In this way, all of the memory allocated to the windows is returned to the system's memory pool and can be used by other programs.

Example Program: Hello World

HELLO.C is an example of how windows and screens can be tied together into a simple program. Type in the program as it appears in Program 7-4. Notice that when there's an error, everything which has been opened up to that point must be closed before exiting. To eliminate a lot of redundant code, we check each item we might open against NULL. If it is NULL, we assume that it either hasn't been opened or couldn't be opened. If it is open, we close it. If it isn't open, we ignore it. There are many ways to handle the problem of closing only those things which you have opened. This is only one such solution.

HELLO.C will open a window and a screen and type some message in the window. Notice that Intuition renders the system gadgets for you. You can resize the window and move it about the screen. Play with setting the different flags for the window and adjusting its initial position and maximum/minimum size. You might try opening a second window on the screen and play with the depth arrangement gadgets. Notice how the program uses the Kernel Move( ) and Text( ) routines. The window will stay open for about ten seconds. If

you need more time, change the maximum count for the time delay.

In the following sections we will show you how you can add a close gadget (the little box in the upper left corner) to the window to allow you to close the window when you please.

Getting Input from the User

In the sections which follow, we will discuss methods of getting information from the user. We'll begin by talking about IntuiMessages, the message system used by Intuition to hand user input to your program. Next, we will explain the simplest of user inputs, a CLOSEWINDOW request. Finally, we'll cover application gadgets and menus.

IDCMPs and CLOSEWINDOW Gadget

As you have already learned, the system uses gadgets to close, resize, and depth-arrange the windows. However, up to this point, resizing and depth arrangement have been transparent. Obviously, the close gadget cannot be transparent to a program, since a program is responsible for closing all of the windows, screens, and libraries it has opened before exiting. If close were transparent like the other system gadgets, then the program would never be given an opportunity to shut everything down before it disappeared. How, then, does Intuition communicate with the main program?

All of the user input for the Amiga is channeled through a task called *input.device.* This task monitors the keyboard, the mouse, and other input devices that might be supplied with the computer (a light pen, joystick, and so forth) and waits for something to happen. When such an input event occurs, Intuition first checks to see whether it can use the event (if, for instance, the user tries to depth-arrange the windows). If it can't use the information, the event is passed along to the active window, the only window which can be receiving input from the outside world.

To talk to our task, Intuition sends an IntuiMessage. IntuiMessages are sent to a task's IDCMP (Intuition Direct Communication Message Port) which is its hotline to Intuition. Intuition automatically opens a set of IDCMPs for any window that needs to talk directly to Intuition. One of the ports is for sending messages to Intuition; the other is to receive messages from it.

IntuiMessages are a special kind of interprocess communication message type, specifically designed for communicating input events to the various tasks within the computer.

```
struct IntuiMessage {
    struct Message ExecMessage;
    ULONG Class;
    USHORT Code, Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink; }
```

These have the following meanings:

ExecMessage	A Message structure maintained by the Exec Kernel routines. It is used to link this message with others in the system and at this message port.
Class	The bits here correspond to the IDCMP flags specified below.
Code	Contains special information about the particular class of the message. It may contain the raw key code generated by the keyboard, or it may contain information about which menu item was selected.
Qualifier	As with the message's Code, the exact meaning of this field depends on the context in which it was received.
MouseX, MouseY	These are the position of the mouse pointer relative to the upper left corner of the window which received the IntuiMessage.
Seconds, Micros	The system time when the message was sent. You might want to use these values to see if the user has double-clicked something.
IAddress	Points to the object (gadget, screen, and so on) which the message concerns. You will see later how this is used to determine which gadget is providing us with input.
IDCMPWindow	Contains a pointer to the Window structure representing the window to which the message pertains. In this way, we know which window is generating input and can take the appropriate action.
SpecialLink	This field is intended to be used only by the system and generally points to another IntuiMessage.

There are a fixed number of IntuiMessage classes. These correspond to the IDCMP flags you can set when you open a window. You must set these flags in the IDCMP field for the

window if you want to receive a message of that kind. For instance, if you want to receive MOUSEBUTTONS messages, then you must set the MOUSEBUTTONS flag in the IDCMPFlags field of the NewWindow structure when you open the window. Alternatively, you can call the ModifyIDCMP( ) function to modify which IntuiMessages your window will receive.

If you do not set the proper IDCMP flags for the window, then Intuition will "intercept" those messages it believes you don't want to see, and your task will never receive them. Also note that only the active window can receive input from the outside world. If your window is not currently active, then you will receive *no* IntuiMessages (except for disk in/out messages, should you request them). As with the window flags, the IDCMP flags can be divided into groups. The first set of flags relates to mouse input:

MOUSEBUTTONS—Setting this flag causes an IntuiMessage to be sent if either of the mouse buttons has been pressed or released. Intuition refers to the left button as *Select* and the right button as *Menu*. You will receive a message of this class only if the key action doesn't mean anything to Intuition. If Intuition can make use of the input event, then the user's program will never see the event occur. Should you receive an IntuiMessage of this kind, the code field will tell you what has really happened. There are four possibilities: SELECTDOWN, SELECTUP, MENUDOWN, MENUUP.

MOUSEMOVE—This flag tells Intuition to send you a message every time the mouse moves. This can result in a large number of IntuiMessages; thus, it's generally a good idea to deal with them as quickly as possible. The position of the mouse is reported in the MouseX and MouseY fields of the IntuiMessage structure. Note that MouseX and MouseY are relative to the upper left corner of the window to which this message pertains.

DELTAMOVE—This is similar to MOUSEMOVE above except that mouse movements are reported as delta (changes in) movement rather than absolute coordinates. You must specify this flag along with the MOUSEMOVE flag above.

This next set of IDCMP flags relates to gadgets. These might not make sense now, but they should once you've read this entire section. Application gadgets are discussed below. For now, the only important flag is CLOSEWINDOW.

GADGETDOWN—Setting this flag causes Intuition to send an IntuiMessage when your gadget is first selected. This will happen only if the gadget was created with the flag GADGIMMEDIATE set in Activation field of the Gadget structure.

GADGETUP—Causes Intuition to send an IntuiMessage when your gadget is released. For this to happen, you must have set the RELVERIFY flag in the Activation field of the Gadget structure when the gadget was created. This is usually the more important message.

CLOSEWINDOW—Because this is so important, it has been given a special IDCMP flag. When the close box of the window is selected, a message of CLOSEWINDOW class is sent to the window's IDCMP. Note that this will happen only if the CLOSEWINDOW flag has been set in the IDCMPFlags field of the NewWindow structure and the WINDOWCLOSE flag has been set in the Flags field when the window is created.

This last set of flags is more or less miscellaneous:

MENUPICK—If this flag is set, you will receive an IntuiMessage every time the menu button is clicked. If a menu item was selected, then the menu item number will be the Code field of the IntuiMessage structure; otherwise, the Code field will hold the value MENUNULL. We haven't talked about menus yet, but they are discussed in a later section of this chapter.

NEWSIZE—If this flag is set, you will receive a message of this kind whenever the window is resized. You can find out the new size of the window by looking at the Width and Height fields in the Window structure pointed to by the IAddress field in the IntuiMessage.

RAWKEY—If you set this flag, you will receive a message every time a key is pressed on the keyboard. The keycode (not the ASCII equivalent) will be sent in the Code field.

Remember that these are both the IntuiMessage's class and the IDCMP flag relating to that IntuiMessage type.

Now that we know all of this, how do we look for a CLOSEWINDOW? You can modify HELLO.C, as in Program 7-5, to make it end by pressing the close box. Notice that only three things have to be changed. First, we add the declaration of the IntuiMessage variable. Second, we include the IDCMP flag CLOSEWINDOW in the NewWindow IDCMPFLags *and* the WINDOWCLOSE flag in the Flags declaration. And, third, we include the code to wait for a message and close the window if the message is of the right class. Program 7-6 shows how the new HELLO.C can be modified to print out the current mouse position as you move around the screen.

As the examples show, you need only three commands to deal with IntuiMessages: Wait( ), GetMsg( ), and ReplyMsg( ). Wait( ) waits for Intuition to signal that an input event has occurred; GetMsg( ) gives access to the IntuiMessage; ReplyMsg( ) tells Intuition that we have received the IntuiMessage and are ready to receive another. In a later section we will deal with these commands more fully.

It's easy to modify HELLO.C to print other information which is sent through IntuiMessages. You may have noticed that when the HELLO.C window is no longer selected, the IntuiMessages stop coming. Try running the program and deselecting the window by pointing and selecting somewhere in the screen not enclosed by the window. Then move the pointer around the screen. Notice that the cursor position numbers are no longer changing as the mouse is moved around. Only the active window (the window with a solid, not dithered, title bar) is allowed to receive input from the user.

Program 7-4. Open a Window

```
#include <lattice/stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
/*
 * hello.c
 *
 * This program opens a simple screen (using the XOpenScreen function),
 * then it displays some text in a window it opens.  The window and
 * screen close by themselves after a short while.
 */

/* keep the compiler happy about type checking */
struct Screen *XOpenScreen();
struct Window *OpenWindow();

struct NewWindow newwindow = {
   100, 50,       /* LeftEdge, TopEdge */
   300, 100,       /* Width, Height */
   0, 1,          /* DetailPen, BlockPen */
   NULL,          /* IDCMPFlags */
   WINDOWSIZING | SIZEBRIGHT | WINDOWDEPTH | WINDOWDRAG
      | SMART_REFRESH | ACTIVATE | NOCAREREFRESH,   /* Flags */
   NULL,          /* FirstGadget */
   NULL,          /* CheckMark */
   "Sample Window",     /* Title */
   NULL,          /* Screen (value will be filled in) */
   NULL,          /* BitMap */
   180, 50, 640, 200,    /* MinWidth, MinHeight, MaxWidth, MaxHeight */
   CUSTOMSCREEN       /* Type */
```

```
};
struct Window *window;
struct Screen *screen;
struct GfxBase *GfxBase;    /* note: the name MUST be "GfxBase" */
struct IntuitionBase *IntuitionBase;    /* as above */

main ()
{
   int i;
   if ((IntuitionBase =
   (struct IntuitionBase *)OpenLibrary("intuition.library", 1)) == NULL)
      Cleanup("Error: couldn't open intuition library\n");
   if ((GfxBase =
   (struct GfxBase *)OpenLibrary("graphics.library", 1)) == NULL)
      Cleanup("Error: couldn't open graphics library\n");
   if ((screen = XOpenScreen("Sample Screen", HIRES, 2)) == NULL)
      Cleanup("Error: couldn't open screen\n");
   newwindow.Screen = screen;
   if ((window = OpenWindow(&newwindow)) == NULL)
      Cleanup("Error: couldn't open window\n");
   Move(window->RPort, 10, 20);
   Text(window->RPort, "All that.. for this?", 20);
   for (i = 1000000; i; i--);
   Cleanup(NULL);
}

Cleanup(ExitText)
char *ExitText;
{
   if (window) CloseWindow(window);
   if (screen) CloseScreen(screen);
```

```
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    if (ExitText) fprintf(stderr, ExitText);
    exit (!!ExitText);    /* NULL returns 0, all else returns 1 */
}
```

Program 7-5. Open a Window and Allow User to Close It

```
#include <lattice/stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
/*
 * hello2.c
 *
 * This program opens a simple screen (using the XOpenScreen function),
 * then it displays some text in a window it opens.  The window and screen
 * close when you select the close window gadget.
 *
 */
/* keep the compiler happy about types */
struct Screen *XOpenScreen();
struct Window *OpenWindow();
struct IntuiMessage *GetMsg();

struct NewWindow newwindow = {
   100, 50,       /* LeftEdge, TopEdge */
   300, 100,       /* Width, Height */
   0, 1,          /* DetailPen, BlockPen */
   CLOSEWINDOW,       /* IDCMPFlags */
```

```
    WINDOWSIZING | SIZEBRIGHT | WINDOWDEPTH | WINDOWDRAG
       | SMART_REFRESH | ACTIVATE | NOCAREREFRESH
       | WINDOWCLOSE,   /* Flags */
    NULL,          /* FirstGadget */
    NULL,          /* CheckMark */
    "Sample Window",    /* Title */
    NULL,          /* Screen (value will be filled in) */
    NULL,          /* BitMap */
    180, 50, 640, 200,   /* MinWidth, MinHeight, MaxWidth, MaxHeight */
    CUSTOMSCREEN      /* Type */
};
struct Window *window;
struct Screen *screen;
struct GfxBase *GfxBase;   /* note: the name MUST be "GfxBase" */
struct IntuitionBase *IntuitionBase;   /* as above */
struct IntuiMessage *message   /* message from intuition */

main ()
{
    if ((IntuitionBase =
    (struct IntuitionBase *)OpenLibrary("intuition.library", 1)) == NULL)
       Cleanup("Error: couldn't open intuition library\n");
    if ((GfxBase =
    (struct GfxBase *)OpenLibrary("graphics.library", 1)) == NULL)
       Cleanup("Error: couldn't open graphics library\n");
    if ((screen = XOpenScreen("Sample Screen", HIRES, 2)) == NULL)
       Cleanup("Error: couldn't open screen\n");
    newwindow.Screen = screen;
    if ((window = OpenWindow(&newwindow)) == NULL)
       Cleanup("Error: couldn't open window\n");
    Move(window->RPort, 10, 20);
    Text(window->RPort, "All that.. for this?", 20);
```

```
    /* do this loop forever */

    for (;;) {
       /* wait for message to be sent to our task */
       Wait(1<<window->UserPort->mp_SigBit);
       while(message = GetMsg(window->UserPort)) {
          /* check for CLOSEWINDOW message class */
          if (message->Class == CLOSEWINDOW) {
             ReplyMsg(message);
             Cleanup(NULL);
          }
       }
    }
}

Cleanup(ExitText)
char *ExitText;
{
    if (window) CloseWindow(window);
    if (screen) CloseScreen(screen);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    if (ExitText) fprintf(stderr, ExitText);
    exit (!!ExitText);   /* NULL returns 0, all else returns 1 */
}
```

Program 7-6. Open a Window and Print Mouse Position

```
#include <lattice/stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
/*
 * hello3.c
 *
 * This program opens a simple screen (using the XOpenScreen function),
 * then displays some text in a window it opens. The window and screen
 * close when you select the close window gadget. When the window is
 * selected, it displays the current mouse coordinates relative to its
 * upper left corner.
 *
 */
/* keep the compiler happy about types */
struct Screen *XOpenScreen();
struct Window *OpenWindow();
struct IntuiMessage *GetMsg();

struct NewWindow newwindow = {
   100, 50,       /* LeftEdge, TopEdge */
   300, 100,       /* Width, Height */
   0, 1,          /* DetailPen, BlockPen */
   CLOSEWINDOW | MOUSEMOVE,    /* IDCMPFlags */
   WINDOWSIZING | SIZEBRIGHT | WINDOWDEPTH | WINDOWDRAG
      | SMART_REFRESH | ACTIVATE | NOCAREREFRESH
      | WINDOWCLOSE | REPORTMOUSE,    /* Flags */
   NULL,          /* FirstGadget */
   NULL,          /* CheckMark */
   "Sample Window",     /* Title */
```

```
    NULL,          /* Screen (value will be filled in) */
    NULL,          /* BitMap */
    180, 50, 640, 200,   /* MinWidth, MinHeight, MaxWidth, MaxHeight */
    CUSTOMSCREEN       /* Type */
};
struct Window *window;
struct Screen *screen;
struct GfxBase *GfxBase;    /* note: the name MUST be "GfxBase" */
struct IntuitionBase *IntuitionBase;    /* as above */
struct IntuiMessage *message    /* message from intuition */

main ()
{
   char *OutText[30];
   if ((IntuitionBase =
   (struct IntuitionBase *)OpenLibrary("intuition.library", 1)) == NULL)
      Cleanup("Error: couldn't open intuition library\n");
   if ((GfxBase =
   (struct GfxBase *)OpenLibrary("graphics.library", 1)) == NULL)
      Cleanup("Error: couldn't open graphics library\n");
   if ((screen = XOpenScreen("Sample Screen", HIRES, 2)) == NULL)
      Cleanup("Error: couldn't open screen\n");
   newwindow.Screen = screen;
   if ((window = OpenWindow(&newwindow)) == NULL)
      Cleanup("Error: couldn't open window\n");
   Move(window->RPort, 10, 20);
   Text(window->RPort, "All that.. for this?", 20);

   /* do this loop forever */

   for (;;) {
      /* wait for message to be sent to our task */
```

```
        Wait(1<<window->UserPort->mp_SigBit);
        while(message = GetMsg(window->UserPort)) {
           /* check for CLOSEWINDOW message class */
           if (message->Class == CLOSEWINDOW) {
              ReplyMsg(message);
              Cleanup(NULL);
           }
           if (message->Class == MOUSEMOVE) {
              Move(window->RPort, 10, 28);
              sprintf(OutText, "%d,%d  ",
                 message->MouseX, message->MouseY);
              Text(window->RPort,
                 OutText, strlen(OutText));
              ReplyMsg(message);
           }
        }
     }
}

Cleanup(ExitText)
char *ExitText;
{
   if (window) CloseWindow(window);
   if (screen) CloseScreen(screen);
   if (GfxBase) CloseLibrary(GfxBase);
   if (IntuitionBase) CloseLibrary(IntuitionBase);
   if (ExitText) fprintf(stderr, ExitText);
   exit (!!ExitText);   /* NULL returns 0, all else returns 1 */
}
```

Intuition Illustration Data Types

Before we can talk about application gadgets and menus, we need to discuss the three different graphic rendering structures used by Intuition. Amiga documentation refers to these structures as "illustration data types." The first of these structures is designed to render text on the screen. Its basic component is a string, but it also includes fields for position, pen colors, drawing mode, and type font. This IntuiText structure has the following fields:

```
struct IntuiText {
    UBYTE FrontPen, BackPen, DrawMode;
    SHORT LeftEdge, TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText; }
```

where the fields have the following meanings:

FrontPen	The pen to use for drawing the text.
BackPen	This field is not currently being used by Intuition.
DrawMode	There are three possible drawing modes: JAM1—Makes Intuition use the FrontPen color register to draw the character. No change is made to the background. JAM2—Makes Intuition draw both the foreground (character) colors and the background colors on the screen. This would make the text blank out the background rather than writing on top of it. XOR—Causes the background beneath the text to be made its binary complement.
LeftEdge, TopEdge	The starting position of the text. LeftEdge is the number of pixels from the left edge of the graphics element in which the text is being rendered. TopEdge is the number of pixels down from the top. For example, if the text is being drawn in a window, then LeftEdge and TopEdge are relative to the upper left corner of the window.
ITextFont	A pointer to the Font structure you wish to use to render the text, or NULL to use the default Topaz font from Preferences.
IText	A normal C string.
NextText	Points to any other IntuiText structures that should be printed at the same time as this one. If this is the last or only IntuiText structure in the list, then set this field to NULL.

The second Intuition illustration data type is a Border structure. It's generally used to construct a border around the gadget (or anything else), but any line figure can be drawn with this data structure.

```
struct Border {
    SHORT LeftEdge, TopEdge;
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    BYTE Count;
    SHORT *XY;
    struct Border *NextBorder; }
```

where the fields have the following meanings:

LeftEdge, TopEdge, FrontPen, BackPen, DrawMode	The same as in IntuiText. Note, however, that JAM2 is not well-defined for Border structures and that LeftEdge and TopEdge position the origin of the border rendering.
Count	The number of vertices in the Border structure (the number of coordinate pairs in the array pointed to by XY).
XY	Acts as a pointer to an array of coordinate pairs. These coordinate pairs are offsets from the initial LeftEdge and TopEdge fields. The first coordinate pair describes the starting point of the line. Each pair after that is the ending point of that line and the starting point of the next. Both positive and negative offsets from LeftEdge and TopEdge are allowed.
NextBorder	A pointer to the next Border structure in a linked list. If this is the only or last Border structure in a list, then this field should be set to NULL.

Note that many Border structures can share the same XY array. This is possible because the XY coordinate pair array is relative to LeftEdge and TopEdge, not the surface in which they are being drawn. Thus, if you were using a Border structure to represent two gadgets that have the same shape, they can both use the same XY array. The Border structures can be chained so that you can change colors and drawing modes as you are drawing the figure. Again, there is no limitation on the shape that you can draw. You are not limited to just horizontal and vertical lines.

The third and last illustration data type is an Image structure. This structure is used if you want to render a bitmap im-

age on the screen. At the lowest level, the Amiga handles graphics through bitmapped screens. Each pixel is controlled by one or more bits. An eight-color display requires three bits for every pixel. These bits are taken from separate, distinct areas of memory called *planes*, or *rasters*, and are combined to determine the color register of the pixel being displayed. Once the color register is determined, the color in that register is displayed on the screen for that pixel.

The Image structure is the closest you can get to this kind of graphics rendering from within Intuition. Like the IntuiText and Border structures, you specify the LeftEdge and TopEdge of the graphics element you are drawing. However, you must also specify the Width and Height, which determine the size in pixels of the rectangular region you are plotting, and the "depth," or number of bit planes, that your image uses. The image data itself is represented in memory as sequential words. The Width is rounded up to the nearest 16 pixels; thus, a 17-pixel-wide image would require two words per row. The bit planes of the image should appear consecutively in memory. Thus, an image 7 pixels wide, 9 high, and three planes deep would require 27 consecutive words of memory.

In an effort to reduce memory overhead, Intuition supports a special PlanePick feature. Using this option, it is possible to determine which of the screen's bit planes an image's bit fields will appear in. Suppose you construct a simple one-plane image. You can tell Intuition to display your image in bit field 0 (to be displayed as color 1), or in bit fields 1, 3, and 4 (or color 26). To do this, you set the appropriate bits in the PlanePick field: 0×1 for the first example, 0×1a for the second. This reduces memory overhead by reducing the number of identical images you need to store in memory. If you'd like the image to be displayed conventionally (each bit plane in the image corresponds to one bit plane for the screen), you set PlanePick to 0×ff.

When an image is rendered, Intuition also examines the PlaneOnOff field. This field allows you to specify what should happen to the bit planes in which your image is *not* appearing. In the second example above, if you used 0×20 for PlaneOnOff, you would set the fifth bit plane to on and the others which are not loaded with our image to off. Note that this applies only to those planes which are *not* receiving a copy of the bit image. Thus, the setting in PlaneOnOff has no bearing on those bit planes specified in the PlanePick field.

One interesting effect of this structure is that very simple images can be created without any bitmap data at all. Set PlanePick to zero, and put a color value in PlaneOnOff. The bits in PlaneOnOff will direct the appropriate bit planes to be filled when displayed, and you'll have a rectangle of whatever color you like, without specifying an actual image.

An Image structure has the following form:

```
struct Image {
    SHORT LeftEdge, TopEdge, Width, Height, Depth;
    USHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage; }
```

in which the fields have the following definitions:

LeftEdge, TopEdge	Specify the position of the top corner of the Image relative to the object it's being rendered in.
Width, Height	The width and height of the rectangular region which encloses the graphics image being represented by this structure. These fields tell Intuition how to interpret the array pointed to by ImageData.
Depth	The number of bit planes needed to represent the image. Again, this is simply to tell Intuition how to interpret the data pointed by ImageData.
ImageData	A pointer to the actual data representing the graphics image. The data should be an array of USHORT.
PlanePick	Tells Intuition which planes you want the graphics image displayed in. Please see text for a more complete explanation.
PlaneOnOff	Indicates what should be done to the bit planes which aren't being used to display the image. A more complete explanation can be found in the accompanying text.
NextImage	A pointer to the next Image structure. If this is the only or last Image structure in a linked list, it should be set to NULL.

As with the Border structure, Image structures are allowed to share ImageData. This helps reduce the amount of data your program must make available to Intuition.

The three illustration data structures, IntuiText, Border, and Image, are used by Intuition when creating and manipulating gadgets, menus, requesters, and alerts.

Application Gadgets

The Amiga Intuition programmer's manual refers to gadgets as the workhorse of Intuition. This is a good description. Almost all user contact with the Workbench is through gadgets. In this section we will show you how to install your own gadgets. The current version of Intuition supports four kinds of "custom" gadgets. These are as follows:

Boolean	These gadgets are either on or off. You can use them like little switches. The CANCEL and RETRY boxes in system requesters are Boolean gadgets.
Proportional	The slider-style gadgets. They can be either one- or two-dimensional. One-dimensional proportional gadgets are used throughout the Preferences program and are used to select keyboard repeat speed, delay time, and the screen colors. The only two-dimensional example we've seen is the gadget in the center of the first Preferences screen which lets you set the position of the corner of the screen within the display.
String	These gadgets are used to input filenames or text of some kind. When you rename a file from Workbench, the box that pops up with the filename in it is a string gadget.
Integer	Integer gadgets are essentially the same as string gadgets and rely on the same data structures. However, integer gadgets automatically convert the string gadget to an INT value.

These four gadgets can be used in a variety of ways to allow almost any input you might want. Building gadgets might seem very complex, but really it's not. Adding gadgets to your program is just a matter of building the appropriate data structures to support them.

There is one new concept which you need to know about to program gadgets: the select box. The select box is the region of the window which represents the gadget to Intuition. Intuition doesn't really care what the gadget looks like, but this is the region in which the gadget is "alive." If the user clicks anywhere in the select box, the gadget becomes selected. Note that the gadget's image and the gadget's select box are two different things. It's possible to program a gadget whose image is nowhere near its select box.

To prepare a gadget, all you have to do is fill in the Gadget structure detailed below. Note that, unlike the NewWindow and NewScreen structures, you can't reuse this one. Each gadget *must* have its own Gadget structure as long as it is being used.

```
struct Gadget
{
    struct Gadget *NextGadget;
    SHORT LeftEdge, TopEdge, Width, Height;
    USHORT Flags, Activation, GadgetType;
    APTR GadgetRender, SelectRender;
    struct IntuiText *GadgetText;
    LONG MutualExclude;
    APTR SpecialInfo;
    USHORT GadgetId;
    APTR UserData; }
```

The fields have the following meanings:

NextGadget	A pointer to the next gadget to be rendered on the window. The last pointer in the chain should be set to NULL.
LeftEdge, TopEdge, Width, Height	Specify the size and location of the select box for the gadget. The select box is the region in the window where the gadget is active. If the user clicks inside the select box, then the gadget is selected.
Flags	There are several flags which can be selected with a gadget. The first four are GADGHIGHBITS, which determine what kind of highlighting Intuition uses when your gadget is selected: GADGHCOMP—Complements the region inside the gadget's select box. GADGHBOX—Draws a box around the gadget's select box. GADGHIMAGE—Displays an alternate Image or Border. GADGHNONE—Does no highlighting.

The next set of gadget flags tells Intuition what the select box's location and size are relative to. These flags allow gadgets to be associated with certain parts of the window (a certain edge for example). For example, you could make a gadget stay near the bottom right edge of a window even if the window is resized.

GRELBOTTOM—Setting this flag makes TopEdge field relative to the bottom of the window (screen, requester, and so forth) it is being displayed in. If GRELBOTTOM is not specified, TopEdge is relative to the top edge of the window. Note that you'll have to use a negative TopEdge if you use GRELBOTTOM and you want the gadget to appear in the window.

GRELRIGHT—As might be expected, setting this flag makes LeftEdge relative to the right edge of the window rather than the left edge. As with GRELBOTTOM, if you use this flag, you'll have to use a negative LeftEdge to get the gadget to appear in the window.

GRELWIDTH—If you use this flag, the Width field is taken as relative to the width of the window. For example, if the width of the window is 140, and you set Width to −20 (while using GRELWIDTH), then the gadget will be 120 pixels wide.

GRELHEIGHT—The same as GRELWIDTH, except it refers to the gadget's and window's height rather than width.

GADGIMAGE—You must set this flag if you're using Image structures to specify the gadget's graphics rendering.

SELECTED—Setting this flag tells Intuition that the gadget should start out selected. If this flag is not set, the gadget will start out unselected.

GADGDISABLED—Tells Intuition that the gadget is not enabled. To change the state of the gadget after it has been rendered on the screen, you have to use the OnGadget( ) and OffGadget( ) functions.

Activation

These are additional flags which relate to gadgets. The following flags may be used in this field:

TOGGLESELECT—Tells Intuition that the gadget is to toggle on and off each time it is selected. You can find the state of the gadget by examining the gadget flag SELECTED.

GADGIMMEDIATE—Will force Intuition to send an IntuiMessage as soon as the gadget is selected.

RELVERIFY—Tells Intuition that we are interested in receiving IntuiMessages from the gadget only if the user still had the pointer over the gadget when the select button was released. This is a way for the

user to verify that he or she really wants to select the gadget. Thus, if the user selects the gadget and, while holding the button, moves off the gadget and releases the button, you will never hear about it.

STRINGCENTER—Setting this flag tells Intuition that you want strings centered in the string gadgets when they are rendered.

STRINGRIGHT—Causes strings to be right-justified rather than centered. If neither STRINGCENTER nor STRINGRIGHT is specified, the string will be left-justified in the string gadget.

LONGINT—Makes the string gadget into an integer gadget. You must specify an initial value in the gadget's input buffer.

GadgetType — This tells Intuition what kind of gadget is being displayed. You must specify one of the following:

BOOLGADGET—A Boolean gadget.

STRGADGET—A string gadget.

PROPGADGET—A proportional gadget.

GadgetRender — Points to either a Border or Image structure representing the graphics of the gadget. If you don't want any graphic rendering for the gadget, set this field to NULL. If GADGIMAGE is set, GadgetRender is assumed to be pointing at an Image structure; otherwise, it must point to a Border structure.

SelectRender — Points to the same type as the GadgetRender field and represents the graphic of the gadget when it is selected.

GadgetText — If the gadget has text related to it, this points to an IntuiText structure which represents that text. The position fields in the IntuiText structure are relative to the top left corner of the gadget's select box. If the gadget does not have any related text, then this field should be set to NULL.

MutualExclude — Should be considered unimplemented until documentation confirms completion of this feature. For a complete discussion of mutual exclude, please see the section on menus.

SpecialInfo — If the gadget is a proportional, string, or integer type, this points to a special structure which holds the data relevant to that type of gadget. If a Boolean gadget is used, this field is ignored.

GadgetId — A special code to distinguish this gadget from all others in the window. IAddress field of the IntuiMessage points to the Gadget structure of the gadget which sent the IntuiMessage. You use the GadgetId field to identify which gadget it was that sent the IntuiMessage.

UserData — A pointer you can use as you please to associate any information with the gadget you find necessary.

When implementing gadgets in C, it's generally easiest to define the last Gadget structure in the list first, and then chain your way through the declarations and define the first gadget in the list last. This way, none of the pointer references will be undefined.

If you are using a proportional gadget, you must also fill out a PropInfo structure, and point the Gadget structure SpecialInfo field at it.

```
struct PropInfo {
    USHORT Flags;
    USHORT HorizPot, VertPot;
    USHORT HorizBody, VertBody;
    USHORT CWidth, CHeight;
    USHORT HPotRes, VPotRes;
    USHORT LeftBorder, TopBorder; }
```

where the fields have the following meanings:

Flags — These are general-purpose flags:

AUTOKNOB—Set this flag if you want to use the autoknob. This gives you a default image for the knob. You have to point GadgetRender at an empty Image structure. Specifying autoknob will also give you a default border.

FREEHORIZ—Allows the proportional gadget to move horizontally.

FREEVERT—Permits the gadget to move vertically.

KNOBHIT—Set by Intuition whenever the knob is selected by the user.

HorizPot, VertPot — The horizontal and vertical percentages of the current gadget position relative to the full size of the gadget. You should set these to initial values before drawing the gadget for the first time. You look here for the current values of the proportional gadget.

HorizBody, VertBody — The horizontal and vertical sizes of the knob represented as percentages of the full size of the gadget.

	This allows the knob to get proportionally larger and smaller if the gadget is tied to the window size and the window is resized.
CWidth, CHeight	The container's (the border of the gadget's) real width and height. These values are set and maintained by Intuition.
HPotRes, VPotRes	The increments by which the gadget moves when clicked on the side.
LeftBorder, TopBorder	The container's true left and top edge.

Intuition maintains CWidth, CHeight, LeftBorder, and TopBorder. You should put values in all of the other fields before starting up the gadget.

If you're using a string or integer gadget, you have to fill in a StringInfo structure and point SpecialInfo at it. The StringInfo structure is defined as follows:

```
struct StringInfo {
    UBYTE *Buffer, *UndoBuffer;
    SHORT BufferPos, MaxChars, DispPos;
    SHORT UndoPos, NumChars, DispCount;
    SHORT CLeft, CTop;
    struct Layer *LayerPtr;
    LONG LongInt;
    struct KeyMap *AltKeyMap; }
```

You must initialize the following fields:

Buffer	A pointer to the initial string. This also points to the string input by the user after the gadget has been used. Note that the buffer must be large enough to hold the largest string you expect to be entered.
UndoBuffer	An optional pointer to the "undo" string. This is the string which will replace the original if Amiga-Q is pressed. If you set this flag to NULL, the undo feature will not be supported.
BufferPos	The initial position of the cursor within the string pointed to by the buffer.
MaxChars	The maximum number of characters to input. This count also includes the NULL terminator on the string. Thus, if you want to allow ten characters to be entered, MaxChars should be set to 11.
DispPos	Represents the position within the buffer of the first character to be displayed.

Intuition will initialize and maintain the following fields for you:

UndoPos	The character position within the undo buffer.
NumChars	The number of characters being held in the buffer.
DispCount	The number of whole characters which are being displayed in the container.
CLeft, CTop	The left and top positions of the container.
LayerPtr	The layer containing this gadget. We're not going to talk about layers in this chapter; however, the Kernel routines need this pointer to handle the gadget properly. As an Intuition programmer, you don't have to worry about this.
LongInt	The integer value of the string if this gadget is an integer gadget.
AltKeyMap	A pointer to an alternate keyboard mapping table. It is beyond the scope of this chapter to discuss this topic. We refer you to the Amiga documentation.

Boolean gadgets are by far the simplest. You don't need any SpecialInfo structures. You can determine the state of a Boolean gadget by looking at the SELECTED flag or by maintaining your own internal variable.

Now that we've defined all of the gadgets and Intuition has drawn them on the screen the way we want them, how do we get the program to use them? Intuition sends us an IntuiMessage through our IDCMP whenever a gadget has been selected by the user. You can have Intuition send two different kinds of IntuiMessages.

If you set the GADGIMMEDIATE flag in the Activation field of the Gadget structure, then you will receive an IntuiMessage of the class GADGETDOWN whenever the user first selects that gadget. If no other activation flags are set, you will not get any more messages about the gadget until it is selected again.

The other possibility is that you can set the RELVERIFY (RELease VERIFY) flag in the Activation field of the Gadget structure. This tells Intuition to send IntuiMessages only if the pointer was still over the gadget's select box when the select button is released. This gives users a "second chance" if they decide that they really don't want to select that gadget. If a user really does select the gadget, then your program will receive an IntuiMessage of class GADGETUP. If you like, you can set both GADGIMMEDIATE and RELVERIFY, and receive

both GADGETDOWN and GADGETUP IntuiMessages.

Our program now knows that a gadget has been selected (because it has received a GADGETDOWN or GADGETUP IntuiMessage). How do we tell which of our gadgets it is? This is purpose of the GadgetId field in the Gadget structure. Each gadget should be assigned a unique GadgetId number. It's up to the programmer to make sure that this happens. The IAddress field of the IntuiMessage will point to the Gadget structure of the gadget which was selected. Then you must inspect the value of GadgetId to determine which gadget was selected. Finally, you must take whatever action is necessary to deal with the gadget selection.

A program which is handling gadgets might look something like the code fragment below:

```
main( )
{
 struct Window *window;
 struct Message *message;
 struct Gadget *gadget;
 ULONG class;
 USHORT idgadget;
 ... /* initialization code */
 /* main loop of the program */
 for(;;) {
    ...
    Wait(1<<window->UserPort->mp_SigBit);
    /* wait for IntuiMessage */

    /* run through all of the messages */
    while (message =GetMsg(window->UserPort)) {
       /* make local copies of the variables we need */
       class =message->Class;
       gadget =message->IAddress;
       /* give the message back to Intuition */
       ReplyMsg(message);
       switch class {
          GADGETUP:
          GADGETDOWN: idgadget =gadget->GadgetID;
                      handle_gadget(idgadget);
                      break;
          CLOSEWINDOW:  cleanup( );
          ...
       }
    }
 }
}
```

```
            CLOSEWINDOW:  cleanup( );
            ...
        }
    }
  }
}
```

Program 7-7, Mandelbrot, makes extensive use of gadgets. Please refer to it as another example of how gadgets can be handled.

Appendix E lists a few of the Intuition functions which you might find useful in dealing with gadgets. You don't really have to use them if you just open windows with gadgets and close them later. However, if you want to add or remove gadgets after a window has been opened, turn a gadget on or off, or modify the data stored in a proportional gadget, these are the routines you'll need to use to do that.

Menus

Menus offer another way of getting input from the user. A menu bar replaces the screen's title bar when the menu (right) button is pressed. When the user points at a particular name in the menu bar, that menu drops down and the user is allowed to select one of the items from that menu. If a particular item has subitems associated with it, a submenu will appear when the user points at that particular item. The user must then select a subitem from the subitem menu. Intuition only allows menus two levels deep. In other words, a subitem can't have sub-subitems.

Menus are attached to windows, not screens. Each window on a screen may be associated with a different Menu structure. The menu which appears is the menu of the active window.

As with all other Intuition functions, installing menus in your program is no harder than setting up a data structure. The Menu structure defines the menu which replaces the title bar when the menu button is pressed. Each name in the menu bar has its own Menu structure.

```
struct Menu {
    struct Menu *NextMenu;
    SHORT LeftEdge, TopEdge, Width, Height;
    USHORT Flags;
```

```
BYTE *MenuName;
struct MenuItem *FirstItem;
SHORT JazzX, JazzY, BeatX, BeatY; }
```

where:

NextMenu	Points to the next Menu structure in the linked list. The last Menu structure should have this field set to NULL.
LeftEdge, TopEdge, Width, Height	Indicate where the select box of the menu header should be placed (the title of the menu which the user points at to make the entire menu drop down). Intuition ignores any values in TopEdge and Height. It uses the top edge of the title bar and the height of the title bar for these values. LeftEdge is relative to the left edge of the screen, and Width is in pixels. You must set these values such that the menu's select boxes don't overlap.
Flags	Used mostly by Intuition. The two flags which can occupy this field are: MENUENABLED—This flag tells you whether or not the menu is enabled. You can set this flag before you submit the menu to Intuition. Afterwards, you must use the functions OnMenu( ) and OffMenu( ) to change the state of this flag. MIDRAWN—If this flag is set, the menu items are currently being displayed to the user.
MenuName	A pointer to a string which represents the name of the menu. The menu's name is the short strip of text which is printed in the screen's title bar when the menu button is pressed. It is displayed with its left edge at the position specified by the field LeftEdge. Thus, the menu's name will always fall somewhere within its select box.
FirstItem	A pointer to the first MenuItem structure which represents a linked list of the items found in the menu. This structure is described below.
JazzX, JazzY, BeatX, BeatY	For the internal use of Intuition only.

Each menu has options below it. These are called menu *items* and are defined in MenuItem structures. A MenuItem structure is linked to a particular menu through the Menu structure's FirstItem pointer. Each item and subitem in a menu must have one of these structures properly linked to something else.

```
struct MenuItem {
    struct MenuItem *NextItem;
    SHORT LeftEdge, TopEdge, Width, Height;
    USHORT Flags;
    LONG MutualExclude;
    APTR ItemFill, SelectFill;
    BYTE Command;
    struct MenuItem *SubItem;
    USHORT NextSelect; }
```

where the fields are as follows:

NextItem — Points to the next item in the linked list of MenuItem structures. If this is the last item in the linked list, it should be set to NULL.

LeftEdge, TopEdge, Width, Height — Indicate the position and size of the menu item's select box. The LeftEdge is relative to the left edge of the menu, and the TopEdge is relative to the bottom line of the menu bar. As with the Menu structure, you are responsible for insuring that the select boxes don't overlap. If you are using checkmarks, you have to make sure that there's room for the checkmark imagery. For high-resolution (640 wide) screens, Amiga recommends adding CHECKWIDTH to the LeftEdge. If you're using a low-resolution (320 wide) screen, you should add LOWCHECKWIDTH.

Flags — The following flags are shared by your program and Intuition:

CHECKIT—Tells Intuition that you would like this item to be rendered with a checkmark in front of it. This tells Intuition to reserve some space in front of the menu item for the checkmark.

CHECKED—If you set this flag and CHECKIT is set when you first submit the menu strip, the item will be selected. You can check this flag to find out whether it is still checked after the user has played with the menus.

ITEMTEXT—Used to tell Intuition that the structure pointed to by the ItemFill field is IntuiText, not an Image structure.

COMMSEQ—Setting this flag tells Intuition that the menu entry has a command-key alternative as specified in the Command field below.

ITEMENABLED—Tells Intuition whether the item is currently enabled. If it is not enabled, the item will be dithered and the user will not be able to select it.

You must set this flag to its proper state before submitting the menu strip to Intuition. Afterwards, you can change the state of the item with the OnMenu( ) and OffMenu( ) functions.

The following flags indicate how the item is to be highlighted when selected. These are referred to as HIGHFLAGS in the Amiga documentation.

HIGHCOMP—All of the bits in the select box should be complemented.

HIGHBOX—A border should be drawn around the outside of the item's select box.

HIGHIMAGE—Use an alternative image for the selected menu item (see the SelectFill and ItemFill fields).

HIGHNONE—Perform no highlighting.

These last two flags are used solely by Intuition. You can inspect them, but don't change their values.

ISDRAWN—Set by Intuition when the item's subitems are being displayed on the screen.

HIGHITEM—Intuition sets this flag if the item is currently highlighted.

MutualExclude — Refers to those items which are excluded when this one is selected. This is described in more detail below.

ItemFill — A pointer to the data used to render the menu item entry. It can either point to an IntuiText or an Image structure. You must set the ITEMTEXT flag if this points to an IntuiText structure.

SelectFill — Points to the data used to render a selected menu item entry. It must be of the same type as ItemFill.

Command — If the COMMSEQ flag is set in the flags field, this field holds a key the user can press in conjunction with the right Amiga key to simulate selecting this item. This is transparent to the program. As far as your program is concerned, the user has used the mouse to select the option. The menu item will be rendered with the fancy Amiga symbol followed by the proper key. If you are using command keys, you must add COMMWIDTH (for high-resolution screens, LOWCOMMWIDTH for low-resolution screens) to the width of the select box to give them enough room to hold the additional imagery.

SubItem	Should point to the first subitem under this item. If this item doesn't have any subitems, set this pointer to NULL.
NextSelect	After you have processed a menu selection, you should check this field. If this field has something other than MENUNULL in it, there was another menu selection made which you must process.

Menus have one feature which makes them somewhat unique among the input methods available to the Amiga programmer. This is mutual exclude (although gadgets have a mutual exclude field, the current documentation states that this feature is not yet implemented). Mutual exclude refers to the following: Suppose you have a set of options, but only one may be valid at a time (for example, when you cast your ballot, you can only vote for one candidate, not all of them). Mutual exclude makes Intuition handle this situation for you.

Each MenuItem and Menu structure is associated with a number. The first node in each list is zero, the second is one, and so on. For mutual exclude you set the bits which correspond to the MenuItem of those items which you want excluded. For example, suppose you have three items which are mutually exclusive. The first menu item would have the MutualExclude field 0×06 (the binary pattern 110), the second, 0×05 (binary 101), and the third, 0×03 (binary 011).

To complicate things somewhat, Intuition also looks at the CHECKED and CHECKIT flag. The CHECKIT flag tells Intuition whether the menu item is an *action* or *attribute* item. Action items can be selected any number of times and are not rendered with checkmarks. Mutual exclude ignores action items. An attribute item is rendered with a checkmark when selected. Attribute items can be selected only once and must be deselected through mutual exclude. The state of the CHECKED flag tells you whether the menu item is currently selected.

Once you've put together a menu system, you need to install it on the window. This is as simple as making one function call: the SetMenuStrip( ) function will install the menu on a particular window. The only way to change the menu (say, you want to rename one of the items) is to call ClearMenuStrip( ), change the menu system, and then use SetMenuStrip( ) again. This keeps the Intuition routines happy.

Once you know how to install and remove menus, the

next step is using them. Intuition will send an IntuiMessage of class MENUPICK whenever the user presses the menu button. The Code field of the IntuiMessage will hold a "menu number." If the user did not select a menu item, this field will hold the constant MENUNULL. Otherwise, the Code field should be treated as a bit field structure which holds the number of the menu, item, and subitem that was selected. You should use the macros MENUNUM( ), ITEMNUM( ), and SUBNUM( ) to get the menu number, item number, and subitem number rather than manipulate the code directly. This will insure upward source code compatibility should the bit field structure change. Remember, the menus, items, and subitems are numbered starting at zero with the first element in their respective linked lists. You can also use the function ItemAddress( ) to get a pointer to the MenuItem associated with that particular Code.

Once you've processed MENUPICK IntuiMessage, you should check the NextSelect field of the MenuItem structure to make sure that the user didn't select multiple options in one IntuiMessage. NextSelect will hold MENUNULL if no other item was selected, or it will hold the menu number (the same type of number returned in the Code field of the IntuiMessage) of the next selected item. The following code fragment illustrates how to handle this situation.

```
while (menu_number != MENUNULL) {
 /* handle the menu event */
 menu_number = ItemAddress(menu_strip,menu_number)
                    ->NextSelect;
}
```

Multitasking

Your Amiga is a multitasking computer. This is a feature which makes your computer stand out among the personal computer crowd. But what exactly does the term *multitasking* mean? A task is essentially a program which is doing something in the computer. For instance, you may have noticed that every time you pop a disk into the computer, the drive runs for a while and then stops. That's the disk validator running in the background under AmigaDOS. When you pop in the disk, the program is informed (by another task) that a disk has been inserted, and the validator runs the drive to see what

the new disk is about. As you can see, multitasking is a very powerful feature and is easy to get used to. However, with this power comes added complexity and programmer responsibility.

Memory Management and Shared Resources

Each task running in the Amiga thinks that it has complete possession of the computer. In other words, as far as a task is concerned, there are no other tasks. To keep all of the different tasks in the Amiga from running into one another, you have to follow certain conventions when dealing with *shared resources.* Shared resources are those parts of the computer which the various tasks must share with one another because of the resource's own limitations. In the Amiga these resources include the printer, disk drives, memory, and the microprocessor. Although it may seem otherwise because of the speed with which individual instructions are carried out, the Amiga can do only one thing at a time. In the same way, only one task can use a particular location of memory at a time. Only one task can use the printer at a time. Only one task can load or store data on a particular disk drive at a time.

Memory is probably the most troublesome of the shared resources. For example, imagine the chaos if your word processing task started to store the text you were working on in the same memory as your compiler was storing its variables. Since the Amiga has no hardware method of protecting one task's memory from another task's (it has no hardware Memory Management Unit), you have to rely on the software to take care of that for you.

The Amiga Kernel routines provide two functions which are used to allocate memory (make available to a program) and deallocate memory (return to the system). They are AllocMem( ) and FreeMem( ). Note that the Amiga Kernel does *not* keep track of which task has allocated a particular chunk of memory; it's up to the particular task to free up any memory it's allocated before exiting. If you use these routines when you need space for dynamic variables, then there won't be any problems. Note that the standard C functions to allocate memory, for example, malloc( ), rely on these routines. You can use them, too, if you wish. However, as the Lattice C manual points out, when you start with one method, it's probably best to use it consistently throughout your program. That will avoid confusion on your part and probably speed the debugging process.

Intuition also provides a method of allocation and freeing memory through the two routines AllocRemember( ) and FreeRemember( ). AllocRemember( ) will maintain a linked list of all the memory allocated through calls to it. When you want to free the entire chain, you call FreeRemember( ). This offers a convenient way to release memory in the event of a premature program abort.

There are two things which are important to know about memory management on the Amiga. First, the Amiga is very much a pessimist in regards to memory allocation. If the Amiga's internal routines detect a memory management error (one of your programs went wild and stored something where it shouldn't have), then the system declares a "software failure" and resets. Second, if the system runs out of memory, it will probably crash (and may or may not reset, depending on the severity of the crash). This problem is software dependent. One can hope that later versions of the Amiga operating system will be able to cope with the problem of running out of memory in a more elegant fashion.

Interprocess Communication

As we've said above, each process considers itself the sole owner of the computer. But if each task is independent in this way, how does one task communicate with another (as in the example of the disk validator)? This is a problem in any multitasking computer and there are many solutions. The Amiga uses a message dispatcher to channel messages between the tasks. This is much like a telephone switchboard transferring calls between houses. Each house is analogous to a task, and the switchboard is like the message dispatcher.

In the Amiga, messages are queued (that is to say, they are buffered) as they are sent from one task to another, and they are sent to a task one at a time. The message dispatcher deposits the message in the task's message port. It is generally your responsibility to prepare a message port for the task if you expect to be receiving messages from other tasks. The IntuiMessages and IDCMPs that we talked about previously are simply extensions of standard messages and message ports. The commands which you use here are the same commands which we mentioned briefly when we discussed IDCMPs above.

There are two ways a task can check to see if a message

has arrived. The first is to poll its message port until something is found. Although this sounds like a good idea, it's very unfair to the other tasks running in the system. As your task is looping, waiting for input (and doing nothing else), the other tasks aren't given much time to run themselves. It's important to remember that your program is part of a community of programs sharing the same resources. The more resources (such as CPU time or memory) your process hogs, the less the other tasks can get. Thus, the better method for checking for messages is to wait for them, using the Amiga's Wait( ) command. In this method, your task goes to sleep while waiting for a message, and the other tasks are allowed to use the computer.

The Amiga's Kernel routines provide you with two methods to wait for a message. The first way is simple: You just wait for messages to appear at a particular message port. Suppose, however, that you wanted to wait for messages at more than one message port at the same time. This is where the second method, signal bits, comes in.

Each message port is assigned a signal bit. If a message appears at the port, your task is signaled with that signal bit. Signal bits are allocated on a per-task basis, and each task is allowed up to 32 signal bits; however, half of them are allocated for system use, so each task really has only 16 to play with. When you open a window which has IDCMP flags set, Intuition allocates a signal bit for you. If you are waiting for signals from many IDCMPs, all you do is bitwise OR them together and use the Wait( ) function.

You'll find the following commands useful in dealing with interprocess messages:

Wait( ) — As the name implies, this function is used to wait for something to happen. In particular, it waits for a set of signal bits. The task which calls Wait( ) is put to sleep until any one of the signal bits it is waiting for is received. When a task is put to sleep, other lower priority tasks in the system are allowed to run until it is awakened. Execution of this task resumes as soon as a message is received. If there is already a signal bit waiting to be processed, this function returns immediately. Once you're running again, it's the program's responsibility to figure out what sent the signal and to do whatever it needs to do to handle the message.

WaitPort( )	This command is like Wait( ) except that it waits for a message to appear at the specified message port. If a message isn't ready, this routine calls Wait( ) and puts your task to sleep until a message is ready.
PutMsg( )	This command is used to send a message to a particular task's message port. If a task is waiting for a message, then this will reawaken that other task.
GetMsg( )	You use this function to get the message from the message port. GetMsg( ) will return a NULL if no message is ready. Otherwise, GetMsg( ) returns a pointer to a Message structure. This command is used to check for messages in normal situations.
ReplyMsg( )	Once you have a message, you can use this function to tell the sending task that you've received it (this can help keep your multitasking activity somewhat synchronized). When you use this call, you must save your own copy of the message as this function may corrupt the data stored in the message. The use of this command depends on your own particular implementation of interprocess communication. Intuition, for example, requires that you reply to all its messages.

That's essentially all there is to know about interprocess communication on the Amiga. Generally, you will use these commands in connection with IDCMPs. Although it is possible to create subtasks from within a C program (with the Exec AddTask( ) and RemTask( ) functions), you are *not* allowed to do so if the machine is running AmigaDOS. AmigaDOS has its own way of handling tasks, and bypassing these techniques will confuse it to no end.

Fast Floating Point

The floating-point routines that the Lattice compiler uses are not very fast. Lattice C converts all floating-point numbers into double-precision before performing any calculations. The Amiga itself, however, includes the Motorola Fast Floating Point package in an internal math library. We can use these routines to perform floating-point operations when speed is required. In benchmark tests, these routines have outstripped the IEEE-standard Lattice C routines by a factor of ten. There is a penalty for this speed, however. The Lattice C compiler does not know how to use these routines directly, so you have

to use a set of function calls and funny type declarations to make it all work out.

The Lattice compiler uses 32-bit ints, which are conveniently the same size as the Motorola FFP (fast floating point) values. We use ints for all of our communication with the FFP routines. The routines themselves are usually straightforward and easy to use. If we have two FFP variables, *a* and *b*, addition is done by:

```
c = SPAdd(a, b);
```

However, subtraction and division have the operands reversed so that the Motorola equivalent of

```
c = a - b;
```

is

```
c = SPSub(b, a);
```

Other simple arithmetic operations include negation, absolute value, and conversion to and from normal integers. There are also two comparison tests: "test" (the signum function), which returns −1, 0, or 1, depending on the sign of the arguments, and "compare," which returns 1 if its first argument is greater than its second, −1 if less, and 0 if equal. Here is a table of the standard Motorola routines and their C equivalents.

Motorola	**Standard C**
c = SPAdd(a,b);	c = a + b;
c = SPSub(b,a);	c = a − b;
c = SPMul(a,b);	c = a * b;
c = SPDiv(b,a);	c = a / b;
c = SPNeg(a);	c = −a;
c = SPAbs(a);	c = (a > 0) ? a : −a;
i = SPFix(a);	i = (int) a;
c = SPFlt(i);	c = (float) i;
c = SPTst(a);	c = (a > 0) ? 1 : (a < 0) ? −1 : 0;
c = SPCmp(a,b);	c = (a > b) ? 1 : (a < b) ? −1 : 0;

To use these routines it is necessary to open the mathffp library in ROM. We use the normal OpenLibrary( ) call, and assign the value to the integer MathBase. Note that, again, the variable name *must* be MathBase, just as IntuitionBase and GfxBase are required names. So, to use the Motorola routines, add these lines to your code:

```
int MathBase;
MathBase = OpenLibrary("mathffp.library," 0);
if (MathBase == 0) exit(1);
```

The functions described above are only half of the fast floating point library. Two things are missing from it: the ability to convert values back and forth between Motorola and IEEE standard C floats, and the transcendental functions like sine, cosine, and logarithm. These are provided in the mathtrans library, a RAM library, which can be accessed with:

```
int MathTransBase;
MathTransBase = OpenLibrary("mathtrans.library," 0);
if (MathTransBase == 0) exit(1);
```

Two routines are provided for Motorola-to-IEEE conversion. One is SPFieee( ), which converts from IEEE to Motorola; the other is SPTieee( ), which converts back to IEEE. However, here's where the strange data structures become necessary. To pass a parameter to the SPFieee( ) routine, it *must* be an int: A float will be cast to a double and its value hopelessly confused by the time the FFP library gets it. So, we have to assign the value as a float and pass it as an int. The easiest way to do this is with a union:

```
union FFP {int i;float f;
};
```

For example, to convert the value 3.14159 into Motorola format, we'd have to go through the following contortions:

```
union FFP a;
a.f = 3.14159;
a.i = SPFieee(a.i);
```

If we had said SPFieee(3.14159), Lattice C would have converted it to double before passing it. Unfortunately, the SPFieee( ) function expects a float, not a double.

The inverse function is much simpler to use and very helpful. To print the value of our *a* variable above, all that's needed is

```
printf("%f \n", SPTieee(a.i));
```

If you're going to be using this function, you can avoid a few compiler warnings by inserting the line

```
float SPTieee( );
```

at the beginning of your file. The other FFP functions return ints, which is the default type C assumes for functions.

The fast floating point functions (including the transcendental functions) are listed in Appendix I.

A Sample Program: Mandelbrot Sets

Although the concept of mathematical research may be puzzling to some, there is in fact a great deal of time and effort being put into the study of mathematical systems. In recent years there has been a great deal of effort put into an interesting phenomenon called Mandelbrot sets. A Mandelbrot set is a region on the complex plane in which a particular function is divergent when performed recursively on itself. There is no way to predict ahead of time what a Mandelbrot set is going to look like. It is not random, for every time you run the same mathematical function you will get the same set. However, given a certain function, no one can say what it will look like unless it's been seen before.

You don't need to know what this means to appreciate the colorful pictures generated by Mandelbrot set–generating programs. Since the particular function must be applied recursively, it demands a great deal of computer power. You'll have to be patient if you want to generate detailed pictures.

Program 7-7, Mandelbrot.c, uses a menu to change the screen type. You need to have the Image window active to get at the menu. If you want to change the screen resolution and depth, you have to select the options you want from the Screen menu, and then select Restart from the Projects menu. This will clear the screen and start over (bypassing the hello screen) in the new screen configuration. If you have several hours, you might want to generate a 640 × 400, 16-color Mandelbrot set. It's quite an impressive sight.

You might have noticed a mysterious 64-color mode. This is available only on Amigas which have the newer versions of the graphics chips. This mode is called EXTRA_HALFBRITE, which allows the Amiga to use six bit planes and display 64 colors on the screen simultaneously. If you don't notice any difference between 32- and 64-color modes, then your computer doesn't have the newer hardware. Amiga hasn't yet announced a hardware upgrade policy.

There are two ways to exit the program. You can either select the Quit Item in the Project menu, or you can select the close gadget on the image window. You may notice a slight

delay since the IDCMP is not polled very often. This delay shouldn't be more than a second or so at the most.

We've left the program with lots of room for improvement. You could add zoom and relocation gadgets to the gadget window. In addition you might also want to include a way of changing the colors. The Save routine uses an entire byte to hold each pixel (resulting in *huge* files); you might consider a simple compaction technique to reduce file size. In any event, Mandelbrot.c is a good base for lots of Amiga C coding.

More Information

The following books are considered good C teaching guides:

C Primer Plus, by Mitchell Waite, Stephen Prata, and Donald Martin. Howard W. Sams and Company, 1984.

The C Programmer's Handbook, by Thom Hogan. Brady Communications Company, 1984.

The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, the creators of the C programming language. Prentice-Hall, 1978.

From BASIC to C, by Harley Templeton. COMPUTE! Publications, 1986.

The Amiga developer's package includes the following books, all from Commodore-Amiga, Inc.:

Lattice C Compiler Manual.

AmigaDOS User's Manual, AmigaDOS Developer's Manual, and *AmigaDOS Technical Reference Manual.* Originally by Tim King, revised by Jessica King.

Intuition: The Amiga User Interface, by Robert J. Mical and Susan Deyl.

ROM Kernel Manual.

Amiga Hardware Manual, by Robert Peck, Susan Deyl, and J. Miner.

For those of you with access to a major computer network, there are two information sources you might find useful. ARPA Internet users can request to be included in the distribution of the "info-amiga" mailing list by mailing to "info-amiga-request@red.rutgers.edu." Usenet users have access to the newsgroup "net.micro.amiga."

The authors can be reached at these electronic addresses:

Chris Metcalf
metcalf@yale.ARPA
...!decvax!yale!metcalf
metcalf@yalecs.BITNET

Marc Sugiyama
sugiyama@ingres.berkeley.edu
...!ucbvax!ingres.berkeley.edu!sugiyama
sugiyama@fourcc.bitnet

Program 7-7. Mandelbrot.c

```
/*
 * mandelbrot.c
 *
 * A demo program using many of the features of the Commodore Amiga.
 *
 * Chris Metcalf
 * Jan 7, 1986
 */
#include <exec/types.h>
#include <intuition/intuition.h>
#include <lattice/stdio.h>

/* Tell C what these functions are so we aren't given too many warnings */
extern struct IntuiMessage *GetMsg();
extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();
extern struct MenuItem *ItemAddress();

/* define the global variables required by the Amiga library */
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
int MathBase, MathTransBase;

/* Our only global data structure... the eighty column font */
struct TextAttr Font80 = {
   "topaz.font", TOPAZ_EIGHTY, FS_NORMAL, FPF_ROMFONT
};

/* Define a few defaults for initialization */
```

```
#define BIT_PLANES 4        /* one to six (one to five in 640 columns) */
#define WIDTH 1            /* 1 for 320 mode, 2 for 640 mode */
#define HEIGHT 1          /* 1 for 200 mode, 2 for 400 mode */
#define COUNT 100          /* initial maximum iteration count */
#define MAX_COUNT 500        /* maximum that count can be set to */

/* define our global variables */
struct Screen *screen;        /* the screen we're using */
struct Window *m_window, *g_window;    /* the two windows */
struct Menu *menu_strip = NULL;    /* menu strip for this program */
char filename[31] = "mand.save"; /* global filename for load and save */
int scale[MAX_COUNT];         /* the contouring array */
int W = WIDTH, H = HEIGHT;    /* width and height flags for this screen*/
int bit_planes = BIT_PLANES;    /* the number of bit planes in the screen */
int count = COUNT;          /* the max number of iterations */
int left, right, top, bottom;    /* pixel borders of window */
float x_right, x_left;        /* real borders of the window */
float y_top, y_bottom;
int reset = 0;             /* set to 1 or 2 when we want to redraw */
int first_time = TRUE;        /* first time through open_windows() */
int do_contour = TRUE;        /* recompute contouring array */

/* keep the C compiler happy ... */
void calculate_edges(), handle_messages(), handle_message(), handle_gadget();
void handle_menu(), handle_project(), handle_screen();
void open_libraries(), open_windows(), hello_window();
void gadgets_window(), add_menus(), compute_contour();
void load_screen(), save_screen(), close_windows(), done();

/* the numbers of the different gadgets (for our convenience) */
/* Note: declaring enums on the Lattice compiler brings down the system !! */
```

```
#define LOAD_G    Ø
#define SAVE_G    1
#define FILE_G    2
#define REP_G     3

/* the numbers of the different menu items (again for our convenience) */
#define PROJECT_M    Ø
#define SCREEN_M     1
#define PAUSE_M        Ø
#define RESTART_M    1
#define QUIT_M       2
#define HIRES_M        Ø
#define LACE_M       1
#define COLORS_M     2

/* a typedef to allow us to circumvent Lattice C while using FFP */
typedef union { float f; int i; } FFP;  /* Motorola fast floating point */

/* MAJOR Hack!! Lattice C badly broken on successive constant declarations */
float two() { return 2.Ø; }
float negtwo() { return -2.Ø; }
float one() { return 1.25; }
float negone() { return -1.25; }

main()
{
   float x_mult, y_mult;       /* convert to reals from pixels */
   FFP cr, ci, zr, zi;        /* components of c, z, and z^2 */
   FFP si, ti, tr;           /* temp complex numbers */
   FFP zero, four;           /* constant FFP values */
   int i,j;           /* current pixel location */
```

```
    int k;              /* iterations while waiting */

    open_libraries();
    zero.i = SPFlt(0);
    four.i = SPFlt(4);
    x_right = two();
    x_left = negtwo();
    y_top = one();
    y_bottom = negone();
reset_screen:
    if (do_contour) compute_contour(); /* compute "countouring array" */
    if (reset == 1) close_windows(); /* close them if they're open */
    if (reset < 2) open_windows(320*W, 200*H);
    if (reset == 3) goto done;   /* just loaded the window */
    reset = 0;          /* don't reset again! */
    calculate_edges();
    x_mult = (x_right - x_left) / (right - left);
    y_mult = (y_top - y_bottom) / (bottom - top);
    for (i = bottom; i >= top; i--) for (j = left; j < right; j++) {
       cr.f = (j-left) * x_mult + x_left;
       zr.i = cr.i = SPFieee(cr.i);   /* .i not .f !! */
       ci.f = (bottom-i) * y_mult + y_bottom;
       zi.i = ci.i = SPFieee(ci.i);
       for (k = 1; k < count; k++) {
          if (SPCmp(SPAdd(tr.i = SPMul(zr.i, zr.i),
              ti.i = SPMul(zi.i, zi.i)), four.i) > 0) break;
          si.i = SPMul(zr.i, zi.i);
          zr.i = SPAdd(SPSub(ti.i, tr.i), cr.i);
          zi.i = SPAdd(SPAdd(si.i, si.i), ci.i);
       }
       if (k < count) {
```

```
            SetAPen(m_window->RPort, scale[k]);
            WritePixel(m_window->RPort, j, i);
         }
         handle_messages();
         if (reset) goto reset_screen;
      }
   done:
      reset = 0;
      FOREVER {
         Wait(1 << (m_window->UserPort->mp_SigBit) |
              1 << (g_window->UserPort->mp_SigBit));
         handle_messages();
         if (reset) goto reset_screen;
      }
   }

   void calculate_edges()
   {
      left = m_window->BorderLeft;
      right = m_window->Width - m_window->BorderRight - 1;
      top = m_window->BorderTop;
      bottom = m_window->Height - m_window->BorderBottom - 1;
   }

   void handle_messages()
   {
      struct IntuiMessage *message;
      while (message = GetMsg(m_window->UserPort))
         handle_message(message);
      while (message = GetMsg(g_window->UserPort))
         handle_message(message);
```

```
}

void handle_message(message)
struct IntuiMessage *message;
{
   int class = message->Class;
   int code = message->Code;
   APTR address = message->IAddress;

   ReplyMsg(message);
   switch (class) {
      case CLOSEWINDOW:
         done(NULL);
      case NEWSIZE: {
         int x = m_window->Width - m_window->BorderRight - 1;
         int y = m_window->Height - m_window->BorderBottom -1;
         if (right < x) x = right;
         if (bottom < y) y = bottom;
         SetAPen(m_window->RPort, 0);    /* blank window */
         RectFill(m_window->RPort,left,top,x,y);
         reset = 2;       /* "soft" reset */
         break;
      }
      case GADGETUP:
         handle_gadget((struct Gadget *)address);
         break;
      case MENUPICK:
         handle_menu(code);
         break;
      default:
         done("fatal error: bad IntuiMessage");
```

```
    }
}

void handle_gadget(gadget)
struct Gadget *gadget;
{
    extern int count;

    switch (gadget->GadgetID) {
        case LOAD_G:
            load_screen();
            break;
        case SAVE_G:
            save_screen();
            printf("Mandelbrot data saved as %s\n", filename);
            break;
        case REP_G:
            count =
             ((struct PropInfo *)gadget->SpecialInfo)->HorizPot;
            count = (MAX_COUNT * count) / 0xffff;
            if (count < 2) count = 2;
            break;
        case FILE_G:
            break;   /* don't care, we just want the string */
        default:
            done("fatal error: bad GADGETUP");
    }
}

void handle_menu(menu_number)
int menu_number;
```

```
{
   while (menu_number != MENUNULL) {
      switch (MENUNUM(menu_number)) {
         case PROJECT_M:
            handle_project(menu_number);
            break;
         case SCREEN_M:
            handle_screen(menu_number);
            break;
         default:
            done("fatal error: bad menu number");
      }
      menu_number=ItemAddress(menu_strip, menu_number)->NextSelect;
   }
}

#define TOGGLECHECK(number) \
ItemAddress(menu_strip, number)->MutualExclude ^= (1 << ITEMNUM(number));

void handle_project(menu_number)
int menu_number;
{
   static int run = 1;

   switch (ITEMNUM(menu_number)) {
      case PAUSE_M:
         TOGGLECHECK(menu_number);
         run = !run;
         while (!run) {    /* re-entrant code warning! */
            Wait(1 << m_window->UserPort->mp_SigBit |
                 1 << g_window->UserPort->mp_SigBit);
```

```
            handle_messages();
         }
         break;
      case RESTART_M:
         reset = 1;   /* cold reset .. */
         break;
      case QUIT_M:
         done(NULL);
      default:
         done("fatal error: bad menu item under PROJECT");
   }
}

void handle_screen(menu_number)
int menu_number;
{
   switch (ITEMNUM(menu_number)) {
      case HIRES_M: {
         int item = SHIFTMENU(SCREEN_M) | SHIFTITEM(COLORS_M);
         TOGGLECHECK(menu_number);
         W = 3 - W;   /* maps 1 to 2 and 2 to 1 */
         if (W == 2) {
            OffMenu(m_window, item | SHIFTSUB(4));
            OffMenu(m_window, item | SHIFTSUB(5));
         }
         else {
            OnMenu(m_window, item | SHIFTSUB(4));
            OnMenu(m_window, item | SHIFTSUB(5));
         }
         break;
      }
```

```
        case LACE_M:
           TOGGLECHECK(menu_number);
           H = 3 - H;
           break;
        case COLORS_M: {
           int item = SHIFTMENU(SCREEN_M) | SHIFTITEM(HIRES_M);
           bit_planes = SUBNUM(menu_number) + 1;
           do_contour = TRUE;
           if (bit_planes > 4) OffMenu(m_window, item);
           else OnMenu(m_window, item);
           break;
        }
        default:
           done("fatal error: bad menu item under SCREEN");
     }
  }

  /*============================================================================
        INITIALIZE SCREEN AND MANDELBROT WINDOW
  ============================================================================*/

  void open_windows(width, height)
  int width, height;
  {
     static struct NewScreen new_screen = {
        0,      /* LeftEdge (unimplemented) */
        0,      /* TopEdge */
        0,      /* Width (set in code) */
        0,      /* Height (set in code) */
        0,      /* max number of bit-planes (in code) */
        0,1,      /* screen pen colors */
```

```
    NULL,       /* ViewModes (set in code) */
    CUSTOMSCREEN,    /* type of screen */
    &Font80,    /* associated font */
    "Mandelbrot Sets",    /* title of screen */
    NULL,       /* gadgets for the screen */
    NULL       /* custom bitmap */
};
static struct NewWindow new_window = {
    0, 0, 0, 0, -1, -1, CLOSEWINDOW|NEWSIZE|MENUPICK,
    ACTIVATE|WINDOWSIZING|WINDOWDEPTH|WINDOWCLOSE|WINDOWDRAG|
    SMART_REFRESH, NULL, NULL, "Image", NULL, NULL,
    100, 60, 0, 0, CUSTOMSCREEN
};

new_window.MaxWidth = new_screen.Width = 320*W;
new_window.MaxHeight = new_screen.Height = 200*H;
new_screen.Depth = bit_planes;
new_screen.ViewModes = (W == 2 ? HIRES : 0) | (H == 2 ? LACE : 0);
if (bit_planes == 6) new_screen.ViewModes |= EXTRA_HALFBRITE;
if ((screen = OpenScreen(&new_screen)) == NULL)
    done("couldn't open screen for mandelbrot");
new_window.Width = width;
new_window.Height = height;
new_window.MinWidth = 100*W;
new_window.MinHeight = 60*H;
new_window.Screen = screen;
if (first_time) hello_window(screen);
if ((m_window = OpenWindow(&new_window)) == NULL)
    done("couldn't open window for mandelbrot graphics");
gadgets_window(screen);
}
```

```
#define TEXT(w,x,y,s) { Move(w->RPort,x,y); Text(w->RPort, s, strlen(s)); }
void hello_window(screen)
struct Screen *screen;
{
   int offset = (W == 2) ? 125 : 0;
   static struct NewWindow new_window = {
      35, 72, 250, 66, -1, -1, CLOSEWINDOW,
      WINDOWCLOSE | ACTIVATE | SIMPLE_REFRESH,
      NULL, NULL, NULL, NULL, NULL,
      0, 0, 0, 0, CUSTOMSCREEN
   };
   struct Window *window;

   new_window.Screen = screen;
   new_window.LeftEdge = 35*W;
   new_window.TopEdge = 72*H;
   new_window.Width = 250*W;
   new_window.Height = 66*H;
   if ((window = OpenWindow(&new_window)) == NULL)
      done("couldn't open hello window");
   TEXT(window, 49+offset, 20*H, "CLOSE THE WINDOW TO");
   TEXT(window, 61+offset, 30*H, "START MANDELBROT");
   TEXT(window, 33+offset, 50*H, "Use the menus to change");
   TEXT(window, 69+offset, 60*H, "the parameters");
   Wait(1 << (window->UserPort->mp_SigBit));
   CloseWindow(window);
   first_time = FALSE;
}

/*========================================================================
        INITIALIZE GADGETS WINDOW
========================================================================*/
```

```
void gadgets_window(screen)
struct Screen *screen;
{
   static struct IntuiText load_text = {
      2,2, JAM1,    /* pen colors and mode */
      4,2,       /* starting location relative to select box */
      &Font80, "LOAD", /* text font and string */
      NULL       /* next IntuiText (none) */
   },
   save_text = { 2, 2, JAM1, 4, 2, &Font80, "SAVE", NULL },
   rep_text = { 2, 2, JAM1, 12, 12, &Font80, "Repeat Count", NULL },
   file_text = { 2, 2, JAM1, 24, 12, &Font80, "File Name", NULL };

   /* load and save gadgets */

   static SHORT load_border[5][2] = { 0,0, 41,0, 41,13, 0,13, 0,0 };
   static struct Border load_box = {
      -1, -1,       /* starting position relative to select */
      1, 0, JAM1,    /* pen colors and mode */
      5, (SHORT *)load_border,
      NULL       /* next Border (none) */
   };
   static struct Gadget load_gadget = {
      NULL,       /* next gadget (none) */
      -50,-18,40,12,    /* location and size of select box */
      GADGHCOMP | GRELBOTTOM | GRELRIGHT,    /* flags */
      RELVERIFY,    /* Activation flags */
      BOOLGADGET,    /* type of gadget */
      (APTR)&load_box,    /* pointer to Border */
      NULL,       /* pointer to rendering of selected picture */
      &load_text,    /* associated IntuiText */
```

```
    NULL,       /* mutual exclude bits */
    NULL,       /* special info pointer (none for boolean) */
    LOAD_G,       /* GadgetID */
    NULL       /* UserData pointer */
}, save_gadget = {
    &load_gadget, 10, -18, 40, 12, GADGHCOMP | GRELBOTTOM,
    RELVERIFY, BOOLGADGET, (APTR)&load_box, NULL,
    &save_text, NULL, NULL, SAVE_G, NULL
};

/* file name gadget */

static SHORT file_b[5][2] = { 0,0, 121,0, 121,11, 0,11, 0,0 };
static struct Border file_box =
    { -1, -2, 1, 0, JAM1, 5, (SHORT *)file_b, NULL };
static char undo[31] = "";
static struct StringInfo filestring = {
    filename, undo,       /* string and undo buffers */
    9, sizeof(filename),    /* initial and max cursor pos */
    0,          /* pos of first displayed character */
};
static struct Gadget file_gadget = {
    &save_gadget, 10, 50, 120, 10, GADGHCOMP,
    RELVERIFY, STRGADGET, (APTR)&file_box, NULL,
    &file_text, NULL, (APTR)&filestring, FILE_G, NULL
};

/* repeat count gadget */

static struct PropInfo rep_prop = {
    AUTOKNOB | FREEHORIZ,    /* Flags */
```

```
    0,          /* HorizPot (initialized in code) */
    0,           /* VertPot */
    0xffff / 16,       /* HorizBody */
    0xffff          /* VertBody */
};
static struct Image rep_image;    /* Intuition initializes it */
static struct Gadget rep_gadget = {
    &file_gadget, 10, 20, -20, 10, GADGHCOMP | GRELWIDTH,
    RELVERIFY, PROPGADGET, (APTR)&rep_image, NULL,
    &rep_text, NULL, (APTR)&rep_prop, REP_G, NULL
};

/* new window */

static struct NewWindow new_g_window = {
    170, 30, 140, 95, -1,-1, GADGETUP | MENUPICK,
    WINDOWDEPTH | WINDOWDRAG | SMART_REFRESH,
    &rep_gadget, NULL, "Controls", NULL, NULL,
    0, 0, 0, 0, CUSTOMSCREEN
};

new_g_window.Screen = screen;
new_g_window.LeftEdge = 170*W;
new_g_window.TopEdge = 30*H;
new_g_window.Width = 140*W;
rep_text.LeftEdge = (W == 2) ? 82 : 12;
file_text.LeftEdge = (W == 2) ? 94 : 24;
rep_prop.HorizPot = count * 0xffff / MAX_COUNT;
file_b[1][0] = file_b[2][0] = 1+(file_gadget.Width = (W==2)?260:120);
if ((g_window = OpenWindow(&new_g_window)) == NULL)
    done("couldn't open window for mandelbrot gadgets");
```

```
    add_menus();
}

void add_menus()
{

#define PROJECT_WIDTH   (CHECKWIDTH + COMMWIDTH + 8*8 + 4)
#define SCREEN_WIDTH    (CHECKWIDTH + 6*8 + 4)
#define COLORS_WIDTH    (CHECKWIDTH + 2*8)

#define INTUITEXT(s) { 2,1,JAM1, CHECKWIDTH, 1, &Font80, s, NULL }

   static struct IntuiText
      it_pause = INTUITEXT("Pause"),
      it_restart = INTUITEXT("Restart"),
      it_quit = INTUITEXT("Quit"),
      it_hires = INTUITEXT("Hires"),
      it_lace = INTUITEXT("Lace"),
      it_colors = INTUITEXT("Colors"),
      it_n[6] = { INTUITEXT("64"),INTUITEXT("32"),INTUITEXT("16"),
         INTUITEXT("8"), INTUITEXT("4"), INTUITEXT("2") };

#define N_ITEM(next, top, exclude, text)\
   { next, SCREEN_WIDTH-2, top, COLORS_WIDTH, 10, CHECKIT|ITEMTEXT| \
     ITEMENABLED|HIGHCOMP, exclude, (APTR)text, NULL, NULL, NULL }

   static struct MenuItem mi_n[6] = {
      N_ITEM(NULL, 25, 0x1f, &it_n[0]),
      N_ITEM(&mi_n[0], 15, 0x2f, &it_n[1]),
      N_ITEM(&mi_n[1], 5, 0x37, &it_n[2]),
      N_ITEM(&mi_n[2], -5, 0x3b, &it_n[3]),
```

```
      N_ITEM(&mi_n[3], -15, 0x3d, &it_n[4]),
      N_ITEM(&mi_n[4], -25, 0x3e, &it_n[5])
   };

#define S_ITEM(next, top, flags, text, submenu) \
   { next, 0, top, SCREEN_WIDTH, 10, flags|ITEMTEXT|ITEMENABLED| \
     HIGHCOMP, NULL, (APTR)text, NULL, submenu }

   static struct MenuItem
      mi_colors = S_ITEM(NULL, 20, NULL, &it_colors, NULL),
      mi_lace = S_ITEM(&mi_colors, 10, CHECKIT, &it_lace, NULL),
      mi_hires = S_ITEM(&mi_lace, 0, CHECKIT, &it_hires, NULL);

#define P_ITEM(next, top, flags, text, cmd) \
   { next, 0, top, PROJECT_WIDTH, 10, flags|ITEMTEXT| \
     COMMSEQ|ITEMENABLED|HIGHCOMP, NULL, (APTR)text, NULL, cmd, NULL }

   static struct MenuItem
      mi_quit = P_ITEM(NULL, 20, NULL, &it_quit, 'Q'),
      mi_restart = P_ITEM(&mi_quit, 0, NULL, &it_restart, 'R'),
      mi_pause = P_ITEM(&mi_restart, 10, CHECKIT, &it_pause, 'P');

   static struct Menu screen_m = {
      NULL,        /* NextMenu */
      PROJECT_WIDTH+1,0,SCREEN_WIDTH,0, /* Left,Top,Width,Height */
      MENUENABLED,    /* Flags */
      "Screen",    /* MenuName */
      &mi_hires    /* FirstItem (set in body of function) */
   }, project_m = {
      &screen_m, 1, 0, PROJECT_WIDTH, 0,
      MENUENABLED ,"Project", &mi_pause
```

```
    };
    int i;

    /*
     * Note that we have to reset many of the flags in the body of
     * the code so that when we restore (load) the statically allocated
     * flags will take on the correct values.
     */

    mi_colors.SubItem = &mi_n[5];
    for (i = 0; i < 6; i++) {
       mi_n[i].Flags &= ~CHECKED;
       mi_n[i].Flags |= ITEMENABLED;
    }
    mi_n[6 - bit_planes].Flags |= CHECKED;
    mi_lace.Flags=mi_hires.Flags=ITEMTEXT|ITEMENABLED|CHECKIT|HIGHCOMP;
    mi_hires.MutualExclude = mi_lace.MutualExclude = NULL;
    if (W == 2) {
       mi_n[0].Flags &= ~ITEMENABLED;
       mi_n[1].Flags &= ~ITEMENABLED;
       mi_hires.Flags |= CHECKED;
       mi_hires.MutualExclude = (1 << HIRES_M);
    }
    if (H == 2) {
       mi_lace.Flags |= CHECKED;
       mi_lace.MutualExclude = (1 << LACE_M);
    }
    if (bit_planes > 4) mi_hires.Flags &= ~ITEMENABLED;
    SetMenuStrip(m_window, menu_strip = &project_m);
}
```

```
/*============================================================================
        MISCELLANEOUS FUNCTIONS
============================================================================*/

void compute_contour()
{
   int interval, i, color, j;
   int colors = (1 << bit_planes);
   do_contour = FALSE;
   for (i = interval = 1; ; interval += 2)
      for (color = 1; color < colors; ++color)
         for (j = interval; j; --j, ++i)
            if (i < MAX_COUNT) scale[i] = color;
            else return;
}

void save_screen()
{
   int i,j;
   FILE *fd;

#define PUTW(w) { putc((w >> 8) & BYTEMASK, fd); putc(w & BYTEMASK, fd); }

   if ((fd = fopen(filename, "w")) == NULL) {
      fprintf(stderr, "can't open %s\n", filename);
      DisplayBeep(screen);
      return;
   }
   PUTW(m_window->Width);       /* save the crucial parameters */
   PUTW(m_window->Height);       /* these are saved as SHORTs */
   putc(W, fd);
```

```
    putc(H, fd);
    putc(bit_planes, fd);
    for (i = bottom; i > top; i--) for (j = left; j < right; j++)
       putc(0xff & ReadPixel(m_window->RPort, j, i), fd);
    if (ferror(fd)) {
       fprintf(stderr, "error saving file\n");
       DisplayBeep(screen);
    }
    fclose(fd);
}

void load_screen()
{
    FILE *fd;
    int i, j;
    int width, height;

#define GETW(w) { w = getc(fd) << 8; w += getc(fd); }

    if ((fd = fopen(filename, "r")) == NULL) {
       fprintf(stderr, "can't open %s for read\n", filename);
       DisplayBeep(screen);
       return;
    }
    GETW(width);       /* retrieve the key parameters */
    GETW(height);
    W = getc(fd);
    H = getc(fd);
    bit_planes = getc(fd);
    close_windows();   /* imitate a hard reset */
    open_windows(width, height);
```

```
   calculate_edges();
   for (i = bottom; i > top; i--) for (j = left; j < right; j++) {
      SetAPen(m_window->RPort, getc(fd));
      WritePixel(m_window->RPort, j, i);
   }
   fclose(fd);
   do_contour = TRUE;    /* we'll need to recontour if we restart */
   reset = 3;       /* jump to FOREVER loop when we return */
}

void close_windows()
{
   if (menu_strip) ClearMenuStrip(m_window);
   if (g_window) CloseWindow(g_window);
   if (m_window) CloseWindow(m_window);
   if (screen) CloseScreen(screen);
}

/*========================================================================
                        INITIALIZE AND EXIT CODE
========================================================================*/

void open_libraries()
{
   if ((MathBase = OpenLibrary("mathffp.library", 0)) == NULL)
      done("couldn't open math ffp library");
   if ((MathTransBase = OpenLibrary("mathtrans.library", 0)) == NULL)
      done("couldn't open math transcendental library");
   if ((IntuitionBase = (struct IntuitionBase *)
       OpenLibrary("intuition.library",0)) == NULL)
      done("couldn't open intuition library");
```

```
    if ((GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library",0)) == NULL)
       done("couldn't open graphics library");
}

void done(s)
{
   close_windows();
   if (GfxBase) CloseLibrary(GfxBase);
   if (IntuitionBase) CloseLibrary(IntuitionBase);
   if (MathTransBase) CloseLibrary(MathTransBase);
   if (MathBase) CloseLibrary(MathBase);
   if (s) fprintf(stderr, "mandelbrot: %s\n", s);
   exit (s == NULL ? 0 : 1);
}
```

Chapter 8

Machine Language

Tim Victor

Machine Language

Tim Victor

A number of different devices cooperate to run the Amiga personal computer. For instance, a 6500 microprocessor is used just to read the keyboard; the graphics coprocessor is another complete, if rudimentary, microprocessor. Other peripheral devices also have their own intelligent controllers. But the most powerful device is the central processing unit (CPU), a Motorola 68000 microprocessor, which controls everything else in the computer. Although low-priced computers have only recently begun featuring this processor, it has been available since 1979, and was the most powerful available microprocessor for several years. Its performance has since been surpassed by several devices—including more recent offerings from Motorola—but the 68000 is still an extremely capable processor. The machines that use it are among the most powerful personal computers ever offered.

You can write Amiga programs in a number of languages, including Amiga BASIC, C, Pascal, Modula-2, and assembly language. But the 68000 can only execute programs which are stored as raw numbers in memory chips. To get from programming languages—which use words, numbers, and alphanumeric symbols for human convenience—to machine language, the 68000's native tongue, requires a translator. Most often, the 68000 performs this translation itself by executing a machine language program designed for the purpose. Amiga BASIC, for instance, is an *interpreted* language; the computer translates each BASIC statement into machine code while the BASIC program is running. Other languages (C is one) are translated before execution in a separate process called compilation. The end product in both cases is a series of 68000 *opcodes* which the 68000 can execute directly.

68000 Overview

Why program in machine language? In theory, at least, all computing languages are functionally equivalent: If a program can be written in one language, it can be written in any other language. If that's true, why learn how to program at the machine level? Perhaps the primary reason is efficiency. A program that works well in machine language may be unacceptably

slow when written in a higher-level language. Every high-level language includes some inefficiency, or *overhead,* compared to well-written machine language. The amount of overhead depends on several factors (which language is used, the efficiency of the compiler or interpreter, the cleverness of the programmer, and so on), but it's always present. In the highly competitive commercial program market, the penalties in speed and program size that result from high-level programming can't be ignored. As a general rule, the more speed-critical the task, the more you stand to gain by programming in machine language.

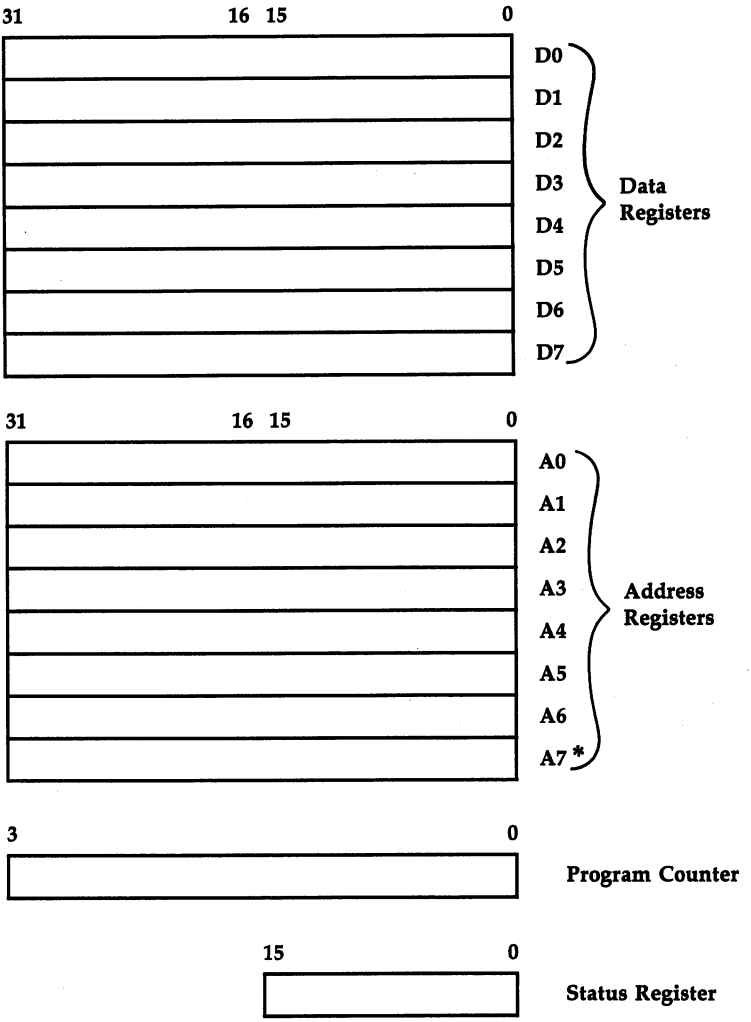
Machine language also offers the ultimate in flexibility and control. High-level languages (particularly Amiga BASIC) have certain built-in constraints that may be inconvenient or impossible to circumvent in some cases. When you can control the processor directly, the only limits are those imposed by the machine itself. Apart from practical considerations, some programmers enjoy the feeling of being in complete control of the computer—for the same reasons that other people like to grow their own vegetables or repair their own cars.

Besides, 68000 machine language isn't particularly hard to learn. If you learned ML programming on a simpler microprocessor like the 6502, the 68000's extensive instruction set may seem almost like a high-level language. Many high-level operations can be performed with just one or two 68000 instructions.

68000 registers. In order to write effective machine language programs, you must understand a few facts about the 68000 microprocessor's internal structure. The 68000 chip contains 17 data storage locations, called *registers* (see Figure 8-1). Sixteen of the 17 onboard registers can be accessed at any given time. Each register can hold a four-byte (32-bit) number, called a *long word.* The 68000 can move data from one register to another, from a memory location to a register, from a register to memory, or from one memory location to another. Most arithmetic and logical operations require that you put one of the necessary operands in a register. The 68000 can also interpret the contents of a register as a memory address and access the memory location specified by a register.

The 68000's internal registers are divided into two groups—address registers and data registers. Each register in a group is numbered 0–7: The data registers are numbered D0–D7 and the address registers are numbered A0–A7. How-

Figure 8-1. 68000 Registers



*** Register A7 ordinarily serves as a stack pointer.**

ever, the seventh address register (A7) is special. Register A7 serves as the *stack pointer,* which points to an area of memory where certain opcodes and addressing modes store and retrieve data. The microprocessor's stack can be located at any even-numbered (*word-aligned*) memory address.

The 68000 actually maintains two different stack pointers, one for each of two operating modes. When the microprocessor is in *user mode*, operations on register A7 refer to the user stack pointer. In *supervisor mode*, register A7 is actually the supervisor stack pointer. This explains how two groups of 8 registers can contain 17 registers. Since the Amiga operates in user mode most of the time, this point isn't too critical.

One of the most distinctive features of 68000 register usage is its generality. Though the registers are divided into a data group and an address group, most operations can be performed on either type of register. There are even addressing modes which use the contents of a data register as an address. This is understandable, since data and addresses have very similar forms, but each group of registers is better suited to its own purpose.

Data types. Although each internal 68000 register can hold 32 bits (a long word) of data, the processor can also handle data in units of 8-bit bytes or 16-bit words. For instance, the instruction MOVE.W 10000,D0 loads register D0 with the 16-bit value contained in memory locations 10000 and 10001. In this case, location 10000 contains the *more* significant byte of the value, and 10001 contains the *less* significant byte. (Note that this high-byte/low-byte order is the reverse of how most microprocessors handle addresses. Another significant difference involves the MOVE instruction, which stores the value from the left operand into the right operand. For instance, the instruction MOVE.W 10000,D0 moves the left operand value—the contents of locations 10000–10001—into the destination expressed by the right operand—the internal register D0. Equivalent instructions on many other computers work in the opposite direction.)

To take another example, the instruction ADD.L (A0),D0 uses the contents of A0 to define a memory address, then adds the contents of the four bytes beginning at that address to the long word contained in data register 0 (D0). The result goes in D0. The bytes that make up the long word are also stored in order of decreasing significance (the high bytes come before the lower bytes).

Any opcode that ends in .W signals a word-length operation, while the suffix .L denotes a long-word operation, and .B signals a byte-length operation. For instance, the instruction

MOVE.B D0,$22223 stores the lowest byte of register D0 in memory location $22223 (the dollar sign before the number indicates that it's expressed in *hexadecimal,* or base 16 notation). You can use byte-length operations to access any byte in memory, but the same is not true of word-length or long-word-length operations. When the 68000 handles any value longer than a byte, its operands must be located in even-numbered memory locations; this arrangement is called *word alignment.* Note that long-word (32-bit) operands must be word-aligned, but need not be *long*-word-aligned in memory (the address must be divisible by 2, but need not be divisible by 4).

Amiga's macro assembler. The Amiga developer's package includes a 68000 macro assembler program named Assem. Like other machine language programs, Assem is called by entering its name from the AmigaDOS Command Line Interpreter (CLI) prompt. After the program name, you would ordinarily supply the name of a text file containing the source code for the program you want to assemble. To indicate that you want to generate object code, the command should also include the option flag *-o* followed by the name you want to call the resulting object file. Consider this example:

assem program.s -o program.o

This command tells the macro assembler to assemble the contents of the source code file called *program.s* and store the resulting object code in a new file called *program.o.*

The Amiga assembler accepts source files written in standard 68000 assembly language format. A program line can have up to four components: a label, an opcode, a list of operands, and a comment (very few lines actually contain all four). Since the assembler distinguishes one component from another chiefly by its position on a line, these are usually called *fields.* Let's look at each field in turn.

Label field. A *label* is an alphanumeric symbol associated with some value. When a label falls at the beginning of a program line, it is given the address of that instruction. Since an Amiga machine language program is usually written without knowing where in memory it will eventually execute, a label is usually *relocatable*: It is assigned the difference between the address of the instruction that follows and the address of the start of the program. No matter where the program eventually resides when it runs, this offset will be the same. Later, when

the computer loads the program into RAM, it replaces relocatable values with actual addresses.

You can also use the EQU assembler directive to create *absolute* labels that represent important numbers and sizes of data structures. These will not be altered when the program loads. The result of a subtraction operation on two relocatable labels is an absolute value.

The Amiga assembler interprets any symbol that starts in the first (leftmost) column of a program line as a label. A symbol that begins in any other column will be treated as a label only if it ends with a colon (:). Labels can contain up to 30 characters, including uppercase and lowercase letters, numerals, underscores (_), and periods(.). However, the first character in a label can't be a numeral. The assembler distinguishes between uppercase and lowercase letters, and doesn't permit duplicate labels (once a value has been assigned to a label, the same label can't be reassigned in that program).

Opcode field. The second field in a line of assembler source code contains the opcode. This field should be preceded by at least one space or tab character to signal that it's not a label, even if the line has no label. The opcode is usually a *mnemonic*—the symbolic name for an actual 68000 instruction—but it can also be a *pseudo-op*, or assembler directive. A pseudo-op has the same format as a machine language opcode, but instead of creating an ML instruction, it tells the assembler to perform a certain job at assembly time. For instance, the DC (Define Constant) pseudo-op tells the assembler to create data rather than an ML instruction. Like a real 68000 instruction, DC includes a size indicator to signal whether the data should take the form of bytes (.B), words (.W), or long words (.L). Here's a typical use:

```
dc.w 1000
```

This statement tells the assembler to generate a word (two bytes) of data representing the value 1000. The DC pseudo-op lets you insert numeric constants or strings into the object code and is equivalent to an initialized variable in a high-level language. If you include several data items after DC, separated by commas, the assembler creates an array of items of the specified size. If you supply a quoted string after DC.B, the assembler creates a string. For instance, this statement generates the string *Hello, world* followed by a carriage return and a linefeed character:

dc.b 'Hello, world',13,10

The DS pseudo-op creates uninitialized data areas. It is followed by a single number or absolute label that shows how many items of a particular size the data area should hold. For instance, DS.W 1 reserves 2 bytes, enough room for one word of data; DS.B 4 generates a 4-byte data area; and DS.L 100 makes room for 100 long words (400 bytes) of data.

The EQU pseudo-op can assign any value to a label. For example, the following line creates an absolute label named *Dozen* and assigns it the value of 12:

Dozen EQU 12

Another important pseudo-op is XREF. This directive tells the assembler that the following label will be used, *but not defined* in this program. When the label is used, the assembler will attach a note to the object code file indicating that an external reference needs to be resolved later on. It is the programmer's job to provide another file which contains a value for this label. This is done in the second stage of assembly, when you *link* the object code to whatever external modules it needs in order to work.

Operand field. In the third field of a program line, after an opcode or pseudo-op, the assembler expects to find an *operand* (or, in many cases, two operands) for the instruction. The operands can be registers, memory locations, numerical values, or combinations of several different items. For instance, in the line MOVE.W 10000,D0, the operand field consists of two operands: a memory location (10000) and a register (D0). The 68000 permits 12 different addressing modes, but only a few instructions can use all 12 modes. In many cases, labels are substituted for memory locations or numeric values. When it finds a label in the operand field, the assembler replaces the label with the value which you previously defined. For instance, if your program begins with the line BIGNUM EQU 10000, then the line MOVE.W BIGNUM,D0 produces the same code as MOVE.W 10000,D0.

Comment field. Comments can be included at any point in a program; they are signaled by a semicolon (;) or an asterisk (*). The assembler ignores everything to the right of a semicolon, regardless of its line position. If a line begins with an asterisk, the entire line is treated as a remark. A completely blank line is also treated as a comment.

Linking machine language modules. Turning a machine language source program into executable object code is ordinarily a two-stage process. The first step—assembly—produces an object code file, but that code is not quite ready to be executed. The second step—called *linking*—permits one or more separate object files to be combined into a single, executable parcel of object code. The Amiga developer's package includes a program called Alink, which does just this.

Why is linking necessary? The small programs in this chapter don't need anything more in the way of code, but they do need more information. When an Amiga ML program calls a built-in system routine (which may be part of AmigaDOS, Intuition, or some other system library), it knows only the name of the routine. To translate the symbolic name of the routine into a numeric value requires an *external reference* to a value defined in another object file. The linker looks for the tags that the macro assembler attached to its output file. For each external symbol which needs to be defined, the linker tries to find a matching symbol which was defined in another file. (Don't worry if that sounds confusing. We'll explain more about linking later on, when we examine the sample programs.)

A single file named AMIGA.LIB contains definitions for all the Amiga's system routines. In this library, the linker can find values to replace the external labels that we used. For instance, the following command links a file called *program.o* to the objects that it needs, producing an executable file called *program*:

alink program.o to program library lib/amiga.lib

Because AMIGA.LIB is ordinarily found in the LIB subdirectory of the assembler disk, we must refer to it as LIB/AMIGA.LIB.

A useful shortcut. Here's a handy command file that lets you perform both stages of the assembly by issuing a single command from the CLI prompt. Enter and save it as an ordinary text file, using the system editor ED or any other text editor. We've named the file MAKE; of course, you can call it anything you like.

```
.key file/a
c/assem <file>.s -o <file>.o -i include -c W2ØØØØØ
if NOT WARN
c/alink <file>.o TO <file> LIBRARY lib/amiga.lib
endif
```

The MAKE command file expects your source code's filename to end with an .S suffix. If no errors occur during assembly and linking, it produces an executable program in a file of the same name, but without any suffix (the first stage of the assembly also produces an intermediate file ending with the suffix .O). For instance, let's assume you have a source file named PROGRAM.S and have created this command file using the filename MAKE:. Entering the following command at the CLI will produce an executable object file named PROGRAM:

execute make program

Programming Fundamentals

Before you can write elaborate machine language applications, you'll need to know how to do fundamental tasks like printing messages on the screen, receiving input from the person who uses your program, and so on. The example programs in this section illustrate typical solutions to a few such problems. More important, they demonstrate how your own applications can access the multitude of native software routines (usually called *libraries*) contained in the Amiga system. For all but the most rudimentary applications, using system libraries is an absolute necessity. The final example in this section introduces the use of *macro* directives, an advanced assembler feature.

Simple text output. Program 8-1 prints a short message to the screen. Enter the program as listed (use ED or another text editor that produces a plain ASCII text file without any control codes or formatting characters), then assemble it with the MAKE command file shown above.

Program 8-1. Hello

```
*
* Programl: print message on console
*

   XREF  _AbsExecBase ;base address for exec library
   XREF  _LVOOpenLibrary   ;exec function to open a library

   XREF  _LVOOutput         ;DOS functions
   XREF  _LVOWrite

******************** open DOS library ******************

   move.l _AbsExecBase,a6 ;find exec library
   move.l #DOS_Name,al    ;pass string containing name
   clr.l dØ              ;expect any version
   jsr   _LVOOpenLibrary(a6)
   move.l dØ,a6
   tst.l dØ
   beq   Abort           ;OK?

********************* print to stdout ******************

   jsr   _LVOOutput(a6) ;get a file handle, already open
   beq   Abort
   move.l dØ,dl          ;get ready to write to it
   move.l #TestMsg,d2    ;address of a string
   moveq.l #TM_Len,d3     ;length of string
   jsr   _LVOWrite(a6)  ;print it
```

```
************************* leave ************************

Abort:
   clr.l dØ
   rts

DOS_Name:
   dc.b  'dos.library',Ø

TestMsg:
   dc.b  'Hello, world',13,1Ø
TM_Len   EQU   *-TestMsg

   END
```

The source code for Program 8-1 begins with four XREF directives. These identify four values (_AbsExecBase, _LVOOpenLibrary, _LVOOutput, and _LVOWrite) which we'll expect the linker to supply. All of these are found in the file named AMIGA.LIB. By design, the Amiga's software system is infinitely flexible. You can add or remove individual system program libraries on the fly and load them into memory in any order. Since the location of a library can change from one minute to the next, your program can't use a fixed address to refer to a library. Instead, it defines each library entry with an offset from a reference point called the *library base.*

Before you can call a routine, you must open a library with the Exec routine known as OpenLibrary. The OpenLibrary routine returns the base address of a particular library. The only exception to this system is the Exec library itself: The label _AbsExecBase points to its base address (actually stored in memory location 4). When the base address of Exec is known, any other library can be located and opened.

Calling an entry in the newly opened library is fairly straightforward. First, store the library's base address in address register A6. Once this is done, the routine is called with a JSR (Jump to SubRoutine) instruction; the JSR must use *address indirect with displacement* addressing mode, which generates the destination address from the contents of A6. For instance, Program 8-1 contains the following instructions:

```
move.l _AbsExecBase,a6
        .
        .
jsr _LVOOpenLibrary(a6)
```

The first instruction puts the base address of Exec in register A6. The label _LVOOpenLibrary contains the offset of OpenLibrary from the base address of Exec. The JSR instruction combines the base address with the offset to produce an actual destination address. By putting the base address in A6, we also tell OpenLibrary where its current library base is.

Most of the Amiga's library routines expect you to supply one or more parameters. In Program 8-1, the AmigaDOS routine that displays our message must be told which file to write, where to find the text data, and how much data there is. In the Amiga, all this data is passed via the 68000's registers, but we need to know which register gets which piece of data. Some libraries are consistent in their register usage, but

for most routines it's necessary to refer to the Amiga developer's documentation.

DOS routines always receive parameters in data registers, with register D1 holding the first parameter. Intuition's library routines accept parameters in both address and data registers; pointers to data structures are passed in the address registers, beginning with A0. The data registers hold single data fields, beginning with D0. Note that some Intuition functions might not use one or the other group of registers.

When Program 8-1 opens the AmigaDOS library, it passes a string containing the name of the library to OpenLibrary in register D1. This string, called DOS_Name, is created with a DC.B directive; the end of the string is marked with a *null terminator* (zero byte). A MOVE instruction puts the address of this string in D1.

All system routines return a result in register D0. The same is expected of application programs which run from the CLI or the Workbench. When Program 8-1 terminates, it clears D0 to indicate a successful execution. The value that OpenLibrary returns is the base address of the library that has been opened. If this value is zero, it shows that the library could not be opened. Program 8-1 tests for this possibility with a TST.L instruction and branches to Abort if the error occurs.

Standard console output. A program run from the CLI can use the CLI's window for its input and output, or it can open a window of its own. Program 8-1 uses the currently active window (the default output device) to print its message on the command line. It calls the AmigaDOS Output routine to get a pointer to the standard output device—in this case, the CLI window, which is already open. (Since output can be diverted from the default output device—the CLI window—to other devices or entities, the general term *file pointer* is often used to describe this pointer.) No arguments are needed for this call. After transferring the pointer to D1, the program loads D2 and D3 with the address and length of the message to be printed. A call to the AmigaDOS Write routine (with the AmigaDOS library's base address still in A6) actually prints the message.

The TestMsg string in Program 8-1 is created with a DC.B directive; notice, however, that this string doesn't end with a zero. The Write routine can be used to output either text or bi-

nary data (which may include zero bytes), so it needs to know the length of the string in advance. The expression *−*TestMsg* permits us to compute this string's length: The asterisk represents the address of the next byte of object code that the assembler will produce. To calculate the length of the string, we simply subtract the address of the start of the message from the address represented by *. This value is assigned to the label TM_Len with an EQU directive.

Echoing the command line. Program 8-2 prints a message with the same technique shown in Program 8-1, but it gets the text of the message from a different source—the command line itself.

When you run a program from the CLI, the system passes information about the *command tail* (everything typed on the command line after the program's name) to your program. The address of the command tail is put in register A0, the first address register. Register D0 receives the tail's length. Before doing anything else, Program 8-2 saves the contents of these two registers on the stack with a MOVEM.L instruction. It's critical to preserve these two items of data, since you can't use them until the AmigaDOS library is open, but the process of opening the library disrupts the contents of A0 and D0. The operand −(SP) pushes the values onto the stack, using *register indirect with predecrement* addressing mode: Before it stores the data, the microprocessor subtracts the length of the data item from the stack pointer SP (actually register A7). This is the standard means for accessing a stack in 68000 machine language. Since any address register (not just SP) can use this addressing mode, the 68000 can maintain several stacks at the same time. Keep in mind, however, that A7 is the standard stack pointer for many operations, including subroutine calls, traps, and interrupts.

Later in the program, assuming that the AmigaDOS library can be opened successfully, we retrieve the stored contents of A0 and D0 from the stack using a MOVEM.L instruction with *register indirect with postincrement* addressing. This addressing mode is signaled by the expression (SP)+. With a pointer to the standard output file in D1, these registers are moved to D2 and D3. Finally, we call the Write routine to do the actual printing.

Copying console input to console output. Program 8-3 shows how to use the standard input device to read data from the CLI.

Program 8-2. Command Line Echo

```
*
* Program2: Command line echo
*

   XREF  _AbsExecBase    ;base address for exec library
   XREF  _LVOOpenLibrary ;exec function to open a library

   XREF  _LVOOutput      ;DOS functions
   XREF  _LVOWrite

******** push pointers to command line arguments ********

   movem.l a0/d0,-(SP)   ;push 'em

******************** open DOS library ******************

   move.l _AbsExecBase,a6 ;find exec library
   move.l #DOS_Name,a1   ;pass string containing name
   clr.l d0              ;expect any version
   jsr   _LVOOpenLibrary(a6)
   move.l d0,a6
   tst.l d0
   bne   DosOk           ;OK?
   movem.l (SP)+,a0/d0   ;no: fix the stack and quit
   bra   Abort
DosOk:

************** print command line to stdout *************
```

```
    jsr    _LVOOutput(a6) ;get a file handle, already open
    move.l d0,d1          ;get ready to write to it
    beq    Abort
    movem.l (SP)+,a0/d0   ;pop address and length
    move.l a0,d2          ;address of a string
    move.l d0,d3          ;length of string
    jsr    _LVOWrite(a6)  ;print it

************************ leave *************************

Abort:
    clr.l d0
    rts

DOS_Name:
    dc.b  'dos.library',0

    END
```

Program 8-3. Copy Console Input to Console Output

```
*
* Program3: copy console input to console output
*

    XREF  _AbsExecBase   ;base address for exec library
    XREF  _LVOOpenLibrary ;exec function to open a library

    XREF  _LVOInput
    XREF  _LVOOutput     ;DOS functions
```

```
    XREF  _LVORead
    XREF  _LVOWrite

******************** open DOS library ******************

    move.l _AbsExecBase,a6 ;find exec library
    move.l #DOS_Name,a1  ;pass string containing name
    clr.l d0             ;expect any version
    jsr   _LVOOpenLibrary(a6)
    move.l d0,a6
    tst.l d0
    beq   Abort          ;OK?

*************** read a string from console **************

    jsr   _LVOInput(a6)  ;get input file handle, already open
    move.l d0,d1         ;file handle
    beq   Abort
    move.l #MsgBuff,d2   ;buffer address
    moveq.l #80,d3       ;buffer length
    jsr   _LVORead(a6)   ;fill buffer

    move.l d0,MsgLen     ;save length
    cmp.l #1,d0          ;anything there?
    bmi   Abort          ;quit if not

******************** print to stdout *******************

    jsr   _LVOOutput(a6) ;get a file handle, already open
    move.l d0,d1          ;get ready to write to it
    beq   Abort
```

```
    move.l #MsgBuff,d2     ;address of a string
    move.l MsgLen,d3       ;length of string
    jsr   _LVOWrite(a6)    ;print it

************************* leave ************************

Abort:
    clr.l d0
    rts

    SECTION  data,DATA

DOS_Name:
    dc.b  'dos.library',0

    SECTION  mem,BSS

MsgBuff:
    ds.b  80

MsgLen:
    ds.l  1

    END
```

The AmigaDOS Input routine returns a pointer to the standard input device, which is ordinarily the CLI window itself. This pointer is transferred to register D1; the address of an input buffer goes into D2, and the length of the buffer (80, in this case) goes in D3. The Read routine gets as many bytes as it can until it either reaches the end of the file or fills the buffer. Since this input is coming from the console device, the end of the file is signaled by a carriage return at the end of a line. The CLI uses this same routine to get commands, so normal line editing is available.

The SECTION directives in Program 8-3 divide the program into sections according to the type of data that each part contains. Although no SECTION statement precedes the first part of the program, this segment has the type of CODE by default. The second section has the type DATA, since it contains no program code. The final section, with the BSS (Block Storage Section) type, contains only uninitialized data fields. Since the initial values of these fields are unspecified, they don't need to be included in the object file. Only when the program is loaded will space be allocated for block storage. This saves a lot of disk space for programs that handle large arrays or other data entities.

After you divide a program with SECTION, the different program segments no longer need to be adjacent—each segment can be loaded anywhere in memory. When would this be desirable? Frequent allocation and deallocation of memory, especially in a multitasking environment, can leave the system with many small, isolated chunks of free memory, a condition known as *fragmentation*. In this situation, there might not be enough contiguous RAM to hold an entire program (or a large display bitmap, and so on) if the entire program entity has to be in one place. By loading the different sections into separate regions of memory (a technique called *scatter loading*) the Amiga's memory is used more effectively.

Macro instructions. Program 8-4, which types files on the console, introduces an assembler feature known as the *macro instruction*, for which Amiga's macro assembler is named.

Program 8-4. Type a File on the Console

```
*
* Program4: list files as requested
*

   XREF  _AbsExecBase   ;base address for exec library
   XREF  _LVOOpenLibrary ;exec function to open a library

   XREF  _LVOInput
   XREF  _LVOOutput          ;DOS functions
   XREF  _LVOOpen
   XREF  _LVOClose
   XREF  _LVORead
   XREF  _LVOWrite

****************** Read and Write MACROs ****************

WriteFile MACRO ;File,String,Len
   move.l \1,d1          ;file to write
   move.l \2,d2          ;string to write
   move.l \3,d3          ;number of chars to write
   jsr   _LVOWrite(a6)   ;write 'em
   ENDM

ReadFile MACRO  ;File,Buffer,Len
   move.l \1,d1          ;file to read
   move.l \2,d2          ;buffer to fill
   move.l \3,d3          ;maximum number of chars to read
   jsr   _LVORead(a6)    ;read 'em
   ENDM
```

```
******************  open DOS library ******************

Existing: EQU   1005    ;DOS file mode- don't create a new file

   move.l _AbsExecBase,a6 ;find exec library
   move.l #DOS_Name,a1    ;pass string containing name
   clr.l d0               ;expect any version
   jsr    _LVOOpenLibrary(a6)
   move.l d0,a6           ;a6 holds DOS library pointer from now on
   tst.l d0
   beq    Abort           ;OK?

********* find standard input and output files **********

   jsr    _LVOInput(a6)  ;get input file handle, already open
   move.l d0,StdIn       ;hang on to it
   beq    Abort

   jsr    _LVOOutput(a6) ;get output file handle, already open
   move.l d0,StdOut      ;keep it too
   beq    Abort

******************** get name of file ******************

GetFName:
   WriteFile StdOut,#FName_Prompt,#Prompt_Len

   ReadFile  StdIn,#MsgBuff,#80
   cmpi.b #32,MsgBuff
   bcs    Abort           ;quit on null string

******************** try to open file ******************
```

```
    move.l #MsgBuff,a0    ;name of file
    clr.b -1(a0,d0.L)     ;null terminate it

    move.l a0,d1
    move.l #Existing,d2   ;set access mode
    jsr   _LVOOpen(a6)    ;give it a go

    move.l d0,FileHandle
    bne   Display_File    ;did we get one?

    WriteFile StdOut,#NotFound,#NotFoundLen ;express regrets
    bra   GetFName        ;try again

************* read some bytes from the file *************

Display_File:
    ReadFile FileHandle,#MsgBuff,#80 ;fill buffer
    tst.l d0
    beq   End_of_File     ;done?

****** display them on the standard output device *******

    WriteFile StdOut,#MsgBuff,d0
    bra   Display_File

************** close it now that we're done *************

End_of_File:
    move.l FileHandle,d1
    jsr   _LVOClose(a6)
    bra   GetFName          ;do another
```

```
************************** leave ************************

Abort:
   clr.l d0
   rts

   SECTION  data,DATA

DOS_Name:
   dc.b  'dos.library',0

FName_Prompt:
   dc.b  'Name of file: '
Prompt_Len EQU  *-FName_Prompt

NotFound:
   dc.b  'Couldn''t find that file.',13,10
NotFoundLen EQU *-NotFound

   SECTION  mem,BSS

MsgLen   ds.l  1
MsgBuff  ds.b  80 ;read buffer

StdIn    ds.l  1  ;input file handle
StdOut   ds.l  1  ;output file handle

FileHandle:
   ds.l  1        ;file that is being read

   END
```

In Program 8-4, the MACRO pseudo-op defines the labels WriteFile and ReadFile as *macro instructions*. Once you have defined a macro instruction, the assembler replaces any later occurrence of its name with a predefined block of statements known as the *macro body*. Everything in the program text between the MACRO and ENDM pseudo-ops is treated as the macro body. Within this area, a macro can contain 68000 opcodes, label definitions, or even calls to other macros.

The symbols /1, /2, and /3 in this program's macros represent parameters which should follow the name of the macro when it is called. The comments on the first line of each macro explain what parameters the macro expects to get. The macro named WriteFile generates instructions to prepare the D1, D2, and D3 registers for a call to the AmigaDOS Write routine, then generates the JSR instruction that calls the routine. The macro named ReadFile does the same for the AmigaDOS Read routine.

Once Program 8-4 has opened the AmigaDOS library and located the standard Input and Output routines, it uses the WriteFile macro to print a prompt message. The labels FName_Prompt and Prompt_Len are preceded by the pound symbol (#) to generate an immediate addressing mode operand. Since the pointer to the standard Output routine is stored in StdOut, a long-word variable, this operand must use absolute addressing in order to read the memory location. But FName_Prompt is actually the pointer to the string to be printed. This value (not the contents of the memory location it addresses) should go into D2. Likewise, since the Prompt_Len label has been assigned the actual length of the string, it should also be used as an immediate operand.

After using the ReadFile macro to get a line of input from the console, Program 8-4 tests the first character of the input buffer to see if a blank line was entered. If the line is blank, it terminates. Otherwise, it adds a null terminator to the string and tries to open a file with this name by calling the AmigaDOS Open routine. The value Existing tells AmigaDOS that we expect this file to exist already (and *don't* want to create a new file if this one isn't found). If the call to Open succeeds, we enter a loop beginning at Display_File, which reads a bufferful of bytes from the file and writes them to the standard output file. When the Read routine returns zero as the number of bytes read, the program closes the file and jumps back to GetFName to try to list another file.

Translating C Programs into Machine Language

In addition to machine language programming tools, the Amiga development system includes a compiler for the C language. Since C programs are easier to read than machine language, you'll find that many example programs for the Amiga are published in C. In most cases, it's not difficult to write an equivalent machine language program based on the C example. Since C programs use integer arithmetic to perform logical and arithmetic operations, you can usually do the same operation in a single 68000 instruction (though it may be necessary to move one operand into a data register first). The result can then be assigned to a variable with a MOVE instruction.

The Amiga software library routines perform operations on objects like tasks, message ports, messages, windows, and screens. In the C language these objects are implemented as *structures*, which are compound data types composed of any number of *members*. Each member (also called a *field*) can be a numeric variable, another structure, or a pointer to a variable or structure. A structure can express the logical organization of a software object very neatly, but there's no equivalent built-in facility in machine language. Thus, in order to translate structure operations, we must reproduce the actions that a compiled C program performs when accessing a field within a structure.

The label that identifies a structure is treated as a base address; each of the members that belong to that structure is defined as an offset from that base. The first field in a structure always has an offset of zero. The next field begins immediately after the first field; the offset for this (the second) member is simply the length of the first member, and so on. To access a particular field in a structure, you must add that field's offset to the base address of the structure. Here's a simple C program that defines a structure and accesses one of its members.

```
main( ) /* demonstrate C structures */
{
      struct Car {
            char Make[30]
            char Model[30]
            int Year;
      };
      struct Car NewCar;
      NewCar.Year = 1986;
}
```

This program first defines a structure (called Car) consisting of three members: Make, Model, and Year. Then NewCar is defined as an instance of the Car structure, reserving 62 bytes of space for its contents (two 30-byte strings and one 2-byte integer). Finally, we store the value 1986 in the Year field of NewCar. Now let's do the same job in machine language. This program creates an identical data block and assigns the same value to it.

```
* Car structure definition
Make      EQU 0
Model     EQU Make+30
Year      EQU Model+30
CarLen    EQU Year+2
      move.w #1986,NewCar+Year ;store value in newcar.year
      clr.l d0                 ;end of program
      rts
      CNOP 0,2
NewCar    ds.b CarLen          ;make room for a Car structure
```

In this program the pseudo-op CNOP 0,2 tells the assembler to word-align NewCar in memory. The alignment is necessary because the Year field is accessed with a word-length operation.

In many C programs, a variable will be defined as a pointer to a structure, with a statement like this:

```
struct Car *NewCar
```

Instead of reserving 62 bytes for an entire Car-sized structure, the compiler allocates only 4 bytes for NewCar, enough room for a pointer which can hold the address of a block of memory allocated elsewhere in this program (or possibly in another program). To access a field from a Car structure, NewCar must first be assigned with the address of another structure. Then the following instruction could be used:

```
NewCar->Year = 1986;
```

Here's the machine language equivalent:

```
movea.l NewCar,a0     ;put pointer to the structure in register a0
move.w 1986,Year(a0)  ;then store 1986 in the Year field
```

The operand Year(A0) causes the microprocessor to find the field's address by adding the offset Year to the address held in register A0. Note that the value 60 is substituted for the label Year during the assembly process, but this value is not added to a base address until the instruction actually executes.

Standard Amiga header files. When C programmers need to define a structure for an Exec message or an Intuition window, they use a *header file* to use the standard structure definitions instead of building their own. The C statement *#include* tells the compiler to read a header file from disk and process the source code that it contains. In addition to defining standard structures, header files also give names for standard values that library functions might expect. For instance, passing a value of 1005 to the AmigaDOS Open routine tells it to create a new file if the named file is not found. The header file called *libraries/dos.h* defines a symbol (MODE_NEWFILE) with this value, relieving the programmer of some detail work and making the program more readable.

Machine language programs have equivalent header files which define the same structures and standard values. The file *libraries/dos.i* contains information about DOS structures and values in a format that's usable by the Amiga assembler. Here's a typical use of this header file:

INCLUDE "libraries/dos.i"

The INCLUDE pseudo-op is an assembler directive. When the assembler encounters this statement, it processes all of the code in the *libraries/dos.i* header file, then proceeds to the rest of the program.

There is one important difference between header files intended for C programs and those designed for machine language: In a machine language file, every member of the various structures has a name that starts with a two-letter prefix. Each structure has its own unique prefix, which is shared by every member in that structure.

This scheme is necessary because C compilers work somewhat differently from machine language assemblers. In C, the name of a particular member is meaningful only within the enclosing structure; this permits the programmer to reuse the same field name in many different structures. But in assembly language, all labels are *global;* a particular label can be defined only once and always has the same value regardless of context. By adding prefixes to field names, the Amiga header files can distinguish between similarly named members in different structures. For example, in the C header file called *intuition/intuition.h,* both the Window and NewWindow structures have fields named Title. But in the machine language

header file named *intuition/intuition.i*, the first of these fields is called wd_Title, while the second is called nw_Title.

Cross-referencing libraries. Another important difference between C and machine language concerns the handling of libraries—objects external to the program itself. For instance, say that a C program contains this statement:

```
CloseWindow(MyWindow)
```

The effect of this statement is to call a function named CloseWindow( ), passing it a structure named MyWindow. If the current source file doesn't contain a definition for CloseWindow, the C compiler assumes that this function is defined in a separate object module: It simply tags each use of the undefined function in the object module that it creates and assumes that the linker will resolve the external reference later on.

You can accomplish the same task in machine language, but the process requires a bit more work. The source file must include an XREF pseudo-op for every library routine that the program calls. Consider this example:

```
XREF _LVOCloseWindow
```

This statement tells the assembler to expect the otherwise undefined label _LVOCloseWindow to be used somewhere in the program and to tag the object file for this external reference. Note that every external label that defines a library routine must begin with the four characters _LVO. When you convert C programs to machine language source code, remember to add this prefix to all library routines.

Using Intuition libraries. Program 8-5 demonstrates the use of Intuition libraries in a machine language program. It opens a full-featured Intuition window entitled DemoWindow.

The source code for Program 8-5 employs an INCLUDE pseudo-op to access two standard Amiga header files: *exec/types.i* and *intuition/intuition.i*. The first of these defines a number of macros to emulate C data types. None of these macros are used by Program 8-5 directly, but they must be defined for the second header file, which does use them. The *intuition/intuition.i* file defines structures and labels for describing windows, screens, IDCMPs (Intuition Direct Communication Message Ports), gadgets, and many other Intuition objects.

Program 8-5. Open an Intuition Window

```
*
* program5- open an Intuition window
*

   INCLUDE  "exec/types.i"
   INCLUDE  "intuition/intuition.i"

   XREF  _AbsExecBase
   XREF  _LVOOpenLibrary
   XREF  _LVOWait
   XREF  _LVOOpenWindow
   XREF  _LVOCloseWindow

*************** open the Intuition library ***************

   movea.l #IntuitionName,a1 ;ask for 'intuition.library'
   move.l #29,d0             ;version 29 or later
   movea.l _AbsExecBase,a6
   jsr   _LVOOpenLibrary(a6)
   move.l d0,IntuitionLibrary
   beq   Abort

****************** open our simple window ******************

   movea.l #DemoWindow,a0
   movea.l IntuitionLibrary,a6
   jsr   _LVOOpenWindow(a6)
   move.l d0,MyWindow        ;pointer to newly opened window
   beq   Abort
```

```
****************** wait for mouse click *******************

   movea.l MyWindow,aØ      ;where is our IDCMP receive port?
   movea.l wd_UserPort(aØ),aØ ;which of our task's signal bits will be set
   move.b MP_SIGBIT(aØ),dl ; when Intuition send us a message?

   moveq.l #1,dØ            ;convert the number of the signal bit
   lsl.l dl,dØ              ; into a bit mask by shifting

   movea.l _AbsExecBase,a6
   jsr   _LVOWait(a6)       ;now sit tight until this bit is set

******************** close the window ********************

   movea.l MyWindow,aØ
   movea.l IntuitionLibrary,a6
   jsr   _LVOCloseWindow(a6)

******************** quit in a hurry *********************

Abort:
   clr.l dØ
   rts

   SECTION  data,DATA

IntuitionName:
   dc.b  'intuition.library',Ø

MyGadgets EQU WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
MyFeatures EQU SMART_REFRESH!ACTIVATE
MyFlags EQU MyGadgets!MyFeatures
```

```
MyTitle: dc.b  'Demo Window',0 ;the title of our window

***************** a new window structure *****************

DemoWindow dc.w  20          ; LeftEdge:        20
   dc.w  20                  ; TopEdge:         20
   dc.w  300                 ; Width:           300
   dc.w  100                 ; Height:          100
   dc.b  0                   ; DetailPen:       0
   dc.b  1                   ; BlockPen:        1
   dc.l  CLOSEWINDOW         ; IDCMPFlags:      Window Closed
   dc.l  MyFlags             ; Flags:           All Sorts Of
   dc.l  0                   ; *FirstGadget:    Default
   dc.l  0                   ; *CheckMark:      Default
   dc.l  MyTitle             ; *Title:          'Demo Window'
   dc.l  0                   ; *Screen:         Default
   dc.l  0                   ; *BitMap:         Default
   dc.w  64                  ; MinWidth:        64
   dc.w  20                  ; MinHeight:       20
   dc.w  640                 ; MaxWidth:        640
   dc.w  200                 ; MaxHeight:       200
   dc.w  WBENCHSCREEN        ; Type:            WBENCHSCREEN

   SECTION  mem,BSS

IntuitionLibrary:
   ds.l  1                   ;keep address of Intuition here

MyWindow: ds.l 1             ;address of the structure describing our window

   END
```

After opening the Intuition library, the program loads register A0 with a pointer to a NewWindow structure called DemoWindow, then calls OpenWindow to open it and draw it on the Workbench screen. The ACTIVATE flag in MyFlags tells Intuition to make this window the active window as soon as it's opened. A number of other flags are also set to ask Intuition for a close box, a drag bar, a depth gadget, and a sizing gadget. The IDCMP flag CLOSEWINDOW tells Intuition to send a message when the window's close box is selected. A call to the Exec Wait routine suspends execution of the program until this message arrives.

A *message* is a data structure containing information to be passed between different tasks in the Amiga's multitasking environment. Each message is sent and received at a *message port*, another data structure which is said to be *owned* by a particular task. When one task wants to communicate with another, it asks Exec's message handler to move a message from one of its own ports to a port owned by the receiving task. Since Intuition and an executing application are separate tasks, an Intuition window contains pointers to two IDCMPs, each of which includes a standard Exec message port along with some additional fields. One IDCMP, called the UserPort, receives messages from Intuition. WindowPort, the other, can be used to send messages to Intuition.

When one task receives an Exec message from another, the Exec message handler announces the event by setting one of the receiving task's 32 *signal bits*, also known as *semaphores*. The MP_SIGBIT field in a message port data structure indicates which signal bit is associated with the port. To await the receipt of a message by a particular window, a task must first find the UserPort for that window. The MessagePort structure for this IDCMP will contain the number of the signal bit that indicates its activity. This C expression is equivalent to the three machine language instructions that locate the UserPort IDCMP's signal bit in Program 8-5:

MyWindow->UserPort->MP_SIGBIT

To let the Wait routine test several signal bits at once, Exec expects to receive a bit mask rather than the number of a single bit. The mask should have a one bit in the bit position of every semaphore that you want to test and a zero bit in every other bit position. In a C program, you can make this sort

of bit mask by left-shifting the number 1 to the position of the signal bit with this expression:

1 << BitPosition

In machine language, the LSL (Logical Shift Left) instruction performs this same operation. Program 8-5 puts the shift count in register D1 and sets the lowest bit of D0 by loading it with 1. Then it left-shifts D0 to create the mask, puts the base address of the Exec library in A6, and calls Wait. When Wait returns, we know that the window's close box was clicked; so the program closes the window and terminates.

File copying utility. Program 8-6 demonstrates a more complete Amiga application, which is also very useful. It's a copying utility that lets you copy a file or group of files from one disk to another without the disk swapping normally required on single-drive systems.

This program is designed to be run from the CLI, and requires two command line arguments: After the program name (on the same CLI line), enter the name of the file you want to copy and the directory to which the file should be copied (be sure to separate the filename and directory name with a blank space). By using wildcard symbols, you can also copy several files at once. When the asterisk (*) and question mark (?) characters appear in a filename, they are given special meanings.

An asterisk will match any number of characters in a filename. For instance, the name *file** matches *file1, file of mine, filestuff,* and just plain *file*. To allow a file specification to contain more than one asterisk, Program 8-6 must be able to match a character following an asterisk to some character in the filename being tested. For instance, the specification *.* will match only those filenames that contain at least one period. This can cause occasional problems, since a character in a filename might accidentally match the character following the asterisk when it was intended to match the asterisk itself. For example, consider the filename *add*. Even though it ends with a *d*, it doesn't match the specification **d*, because the first *d* in the filename has already matched the only *d* in the specification. Once you're aware of this problem, it's usually simple to avoid.

The question mark wildcard symbol works in a simpler way: It matches any single character. For instance, the

specification *program?* matches files named *program1, programs,* and *program.,* but not a file named *program.* When more than one file in a directory matches an ambiguous file specification, Program 8-6 copies all of them.

Note that the wildcard scheme in Program 8-6 is the same one employed by most pre-Amiga computers. Since AmigaDOS itself uses an entirely different method for pattern matching, legal AmigaDOS filenames can contain both asterisks and question marks. Program 8-6 may not work properly with files containing these characters.

Including AmigaDOS routines. Program 8-6 begins by INCLUDEing two header files (*exec/types.i* and *libraries/dos.i*) to define the *file information block* structure that it uses to search a directory. After declaring all its external references and defining the WriteFile macro, the program then creates a structure called StoredFile. To speed up the copy process and prevent disk swapping on single-drive systems, the program doesn't write any files until it has identified and loaded into memory all the files that match the file specification. These files are stored in a linked list of structures, each of which contains the name and size of a file along with its contents. Exec's AllocMem and FreeMem routines are called to allocate and deallocate memory for each node in the linked list.

After opening the DOS library and locating the standard output file, Program 8-6 parses the command line. After we find the end of the first argument (which is assumed to be a source file specification), the ScanLoop routine scans this argument looking for slash (/) and colon (:) characters, which indicate that the source argument contains a directory name. It also looks for wildcard characters. If a wildcard is found, the loop called WildLoop checks for more directory characters; if one of these appears, the program prints a "Bad Arguments" message and quits, since wildcards aren't permitted in directory names.

When either loop reaches the end of the source argument, the directory pathname and the filename are both copied to buffers and terminated with zero bytes. Then the destination pathname is also located, copied to its own buffer, and terminated with a zero.

If the source field contains no directory name, the program searches the current default directory for filenames. The CurrentDir routine is called to get a *lock* (a structure which de-

scribes a file or directory) for the current directory. When called, this routine expects D1 to contain a lock for a new current directory. Since no new directory is being set, D1 is cleared. Then the lock for the original current directory is stored in both OldDir and DirLock.

If a directory was named, Lock is called to get a lock for this directory. The ACCESS_READ flag requests a nonexclusive lock; since we're just reading this directory, we may as well let other tasks access it at the same time. The lock that DOS returns is stored in DirLock, then CurrentDir is called to find out what the original current directory is. OldDir receives this lock so the program can restore it when the copying is done.

The AmigaDOS routine Examine fills DirItem, a FileInfoBlock structure, with information about the directory that we'll be searching. Then ExNext is called to scan through all the entries in the directory. If an entry turns out to be a file, not a subdirectory, FMatchLoop compares its name to the file specification. Any filename that matches is printed on the console, then AllocMem is asked to supply enough memory for a StoredFile structure to hold the file. The file is opened, read, and closed, and the next file is examined.

The SaveFiles subroutine is called when the entire directory has been searched. It makes the destination directory current, then links through the list of StoredFiles, opening, writing, and closing each one in turn. When every file has been written, SaveFiles returns to the main procedure, which restores the original current directory and exits.

Program 8-6. Copy Utility

```
*
* program6: quick copy with wildcard file selection
*

    INCLUDE "exec/types.i"
    INCLUDE "libraries/dos.i"

    XREF  _AbsExecBase       ;base address for exec library
    XREF  _LVOOpenLibrary    ;exec: open a library
    XREF  _LVOAllocMem       ;      allocate memory block
    XREF  _LVOFreeMem        ;      free memory block

    XREF  _LVOLock           ;DOS functions
    XREF  _LVOUnLock
    XREF  _LVODupLock
    XREF  _LVOCurrentDir
    XREF  _LVOExamine
    XREF  _LVOExNext
    XREF  _LVOOutput
    XREF  _LVOOpen
    XREF  _LVOClose
    XREF  _LVORead
    XREF  _LVOWrite
    XREF  _LVOExit

WriteFile MACRO ;File,String,Len
    move.l \1,d1          ;file to write
    move.l \2,d2          ;string to write
    move.l \3,d3          ;number of chars to write
    jsr   _LVOWrite(a6)   ;write 'em
```

```
   ENDM

* StoredFile structure
NextSF    EQU Ø           ;pointer to next stored file
SFLength  EQU NextSF+4    ;size of the file in this block
SFName    EQU SFLength+4  ;name of the file in this block
StoredFile EQU SFName+3Ø  ;contents of the file start here

   movem.l aØ/dØ,-(SP)   ;save command line

******************** open DOS library ********************

   move.l _AbsExecBase,a6 ;find exec library
   move.l #DOS_Name,a1   ;pass string containing name
   clr.l dØ              ;expect any version
   jsr   _LVOOpenLibrary(a6)
   move.l dØ,a6
   move.l a6,DosLibrary
   beq   Abort1

******************** find output device ******************

   jsr   _LVOOutput(a6) ;get a file handle, already open
   move.l dØ,StdOut
   beq   Abort1

******************** check command line ******************

   movem.l (SP)+,aØ/dØ   ;get command line info back
   move.l dØ,d2
   add.l aØ,dØ           ;calc address of end of cmd line
```

```
    move.l d0,CmdEnd      ;we'll need this later

********************* get source spec ******************

    cmpi.b #34,(a0)       ;leading quote?
    bne    NoQuote
    addq.l #1,a0
    subq.l #1,d2

    moveq.l #34,d1        ;look for trailing quote
    bsr    InString
    bra    ProcSource

NoQuote:
    moveq.l #' ',d1       ;look for trailing space
    bsr    InString

ProcSource:
    subq.l #1,d0
    ble    BadArgs

    lea    1(a0,d0.L),a3  ;start looking for dest here

    clr.l d1              ;index
    clr.l DirLen          ;end of directory spec
ScanLoop:                 ;look for dir
    move.b 0(a0,d1.L),d3
    cmpi.b #':',d3        ;contains colon?
    beq    FoundDir
    cmpi.b #'/',d3        ;contains stroke?
    bne    NoDir
```

```
FoundDir:
   move.l d1,DirLen      ;end of dir = index+1
   addq.l #1,DirLen
NoDir:
   cmpi.b #'*',d3        ;contains asterisk?
   beq    FoundWild
   cmpi.b #'?',d3        ;contains question?
   beq    FoundWild

   addq.l #1,d1
   cmp.l d0,d1           ;done?
   beq    NoWild
   bra    ScanLoop

********* Check for wild cards in directory spec ********

WildLoop:
   move.b 0(a0,d1.L),d3
   cmpi.b #':',d3        ;colon?
   beq    BadArgs
   cmpi.b #'/',d3        ;stroke?
   beq    BadArgs

FoundWild:
   addq.l #1,d1
   cmp.l d0,d1
   bne    WildLoop
NoWild:

   sub.l DirLen,d0       ;calculate length of filename
   move.l d0,FileLen
```

```
********** copy path and file specs to buffers **********

   move.l DirLen,dØ      ;is there a directory path?
   beq    NoPath

   movea.l #DirPath,a1
   bsr    BuffCopy       ;put it in buffer

NoPath:
   move.l FileLen,dØ
   beq    BadArgs        ;have to have something-- at least *

   movea.l #FileSpec,a1
   bsr    BuffCopy       ;put file name in buffer

*************** process destination spec ***************

   movea.l CmdEnd,a1
StartLoop:
   cmpi.b #' ',(a3)      ;find start of the destination spec
   bne FoundStart
   addq.l #1,a3
   cmpa.l a3,a1
   bne    StartLoop
   bra    BadArgs

FoundStart:
   move.l a1,d2
   sub.l a3,d2        ;find length of dest

************ find end of the destination spec ***********
```

```
TrailLoop:
   cmpi.b #32,-1(a3,d2.L) ;kill trailing stuff
   bhi   NoTrail
   subq.l #1,d2
   beq   BadArgs
   bra TrailLoop
NoTrail:

   cmpi.b #34,(a3)    ;leading quote?
   bne   Quoteless

   cmpi.b #34,-1(a3,d2.L) ;trailing quote?
   bne   BadArgs

   addq.l #1,a3       ;adjust for quotes
   subq.l #2,d2
Quoteless:

   move.l d2,DestLen
   move.l d2,dØ
   movea.l a3,aØ
   movea.l #DestSpec,a1
   bsr   BuffCopy     ;save destination path name

*************** initialize list of files ***************

   clr.l FirstFile
   move.l #FirstFile,LastFile

****************** examine directory ******************
```

```
    tst.l DirLen          ;was one specified?
    bne   ChangeDir       ; yes: make it current

    clr.l d1              ; no: fake pointer to lock
    jsr   _LVOCurrentDir(a6) ;lock of current dir returned in d0
    move.l d0,OldDir
    move.l d0,d1
    jsr   _LVODupLock(a6)
    move.l d0,DirLock
    bra   ExamineDir

ChangeDir:
    move.l #DirPath,d1
    move.l #ACCESS_READ,d2
    jsr   _LVOLock(a6)    ;try to find this device
    move.l d0,DirLock
    beq   Abort1          ;got it?

    jsr   _LVOCurrentDir(a6) ;make it current
    move.l d0,OldDir      ;save old directory

**************** get data about directory ***************

ExamineDir:
    move.l DirLock,d1
    move.l #DirItem,d2
    jsr   _LVOExamine(a6) ;get data about dir
    tst.l d0              ;successful?
    beq   Abort

NextFile:
```

```
   move.l DirLock,d1
   move.l #DirItem,d2
   movea.l DosLibrary,a6
   jsr    _LVOExNext(a6) ;get data about next entry

   tst.l d0               ;is there another?
   beq   QCDone           ; no: quit

   tst.l fib_DirEntryType+DirItem ;is dir?
   bpl   NoMatch          ;can't use it

************* does it match the file spec? **************

   movea.l #DirItem+fib_FileName,a0
   movea.l #FileSpec,a1

FMatchLoop:
   move.b (a1),d0         ;get char from FSpec
   cmp.b (a0),d0          ;match?
   beq   CharMatch        ; yes: process match

   cmpi.b #'*',d0         ;* wildcard?
   bne   NotStar
   move.b 1(a1),d1        ;check if we're at the end of * wildcard
   cmp.b (a0),d1
   bne   NotStarEnd
   addq.l #1,a1           ; yes: advance FSpec and process match
   bra   CharMatch

NotStarEnd:
   bra   WildMatch        ; no: process pseudo match
```

```
NotStar:
   cmpi.b #'?',d0       ;? wildcard?
   bne   NoMatch        ; no: failure
   addq.l #1,a1         ; yes: pseudo match

WildMatch:
   tst.b (a0)
   beq   NoMatch        ;not good enough
   addq.l #1,a0
   bra   FMatchLoop

CharMatch:
   tst.b (a0)           ;matched null terminators?
   bne   NotFNEnd       ; no: go on
   bra   Match          ; yes: success

NotFNEnd:
   addq.l #1,a1
   addq.l #1,a0
   bra   FMatchLoop

Match:

*************** print filename to stdout ****************

   suba.l #DirItem+fib_FileName,a0 ;calculate length of name
   move.l a0,FNameLen   ; and save it for later

   WriteFile StdOut,#DirItem+fib_FileName,a0 ;write it
   WriteFile StdOut,#NewLine,#2

***** allocate space to hold file info and contents *****
```

```
* space needed:
*  4-- pointer to next file
*  4-- length of file
*  30-- name of file
*  ??-- contents of file: fib_Size
* total: length of file+38

   move.l DirItem+fib_Size,d0 ;length of file
   addi.l #38,d0          ; +38
   clr.l d1               ; request any kind of RAM
   movea.l _AbsExecBase,a6
   jsr   _LVOAllocMem(a6)

* unsuccessful? call subroutine to write out the files, then retry
   tst.l d0
   bne   AllocOK

   tst.l FirstFile        ;anything to save?
   beq   NoMem            ;no- quit
   bsr   SaveFiles        ;save everything
   bra   Match            ;try again

***************** load the file into RAM ****************

AllocOK:
   movea.l LastFile,a1    ;save old last file
   move.l d0,LastFile     ;establish new last file
   move.l d0,(a1)         ;link to old last file

   move.l d0,a1
   clr.l (a1)+            ;indicate last file
```

```
    move.l DirItem+fib_Size,(a1)+ ;store file size

    move.l FNameLen,d0
    movea.l #DirItem+fib_FileName,a0
    bsr   BuffCopy        ;store file name

    move.l DirLock,d1
    movea.l DosLibrary,a6
    jsr   _LVOCurrentDir(a6) ;make it current

    move.l #DirItem+fib_FileName,d1
    move.l #MODE_OLDFILE,d2
    jsr   _LVOOpen(a6)    ;Open the file

    move.l d0,d1
    move.l d0,FileHandle
    move.l LastFile,d2
    addi.l #StoredFile,d2
    move.l DirItem+fib_Size,d3
    jsr   _LVORead(a6)    ;Read it

    move.l FileHandle,d1
    jsr   _LVOClose(a6)   ;be nice

NoMatch:
    bra   NextFile

*************** clean up house and leave ***************

QCDone:
    bsr   SaveFiles       ;write out the files
```

```
   bsr   RestoreDir      ;restore original directory

   move.l DirLock,d1     ;free lock structure
   jsr   _LVOUnLock(a6)

   bra   Abort1          ;and end

**************** error handling routines ****************

BadArgs:                 ;bad command line arguments
   WriteFile StdOut,#BAMsg,#BALen
   bra   Abort1

NoMem:                   ;file too big to load
   movea.l DosLibrary,a6
   WriteFile StdOut,#NoMemMSG,#NMMLen
   bra   Abort

SaveFailed1:             ;file is open
   move.l FileHandle,d1
   jsr   _LVOClose(a6)   ;close the file

SaveFailed:              ;couldn't write file
   move.l DestLock,d1
   jsr   _LVOUnLock(a6)

SaveAbort:
   WriteFile StdOut,#FailedMSG,#FailedLen

Abort:
   bsr   FreeAll         ;free all memory
   bsr   RestoreDir      ;restore original current directory
```

```
    move.l DirLock,d1
    jsr    _LVOUnLock(a6) ;unlock source directory
Abort1:
    jsr    _LVOExit(a6)   ;flee

********** subroutine to save all the files ***********

SaveFiles:
    tst.l FirstFile
    beq   SaveDone

*************** set destination directory ***************

    bsr   RestoreDir      ;start path at original directory

    move.l #DestSpec,d1
    move.l #ACCESS_READ,d2
    jsr    _LVOLock(a6)   ;try to find this device
    move.l d0,d1
    move.l d1,DestLock
    beq   SaveAbort       ;got it?

    jsr   _LVOCurrentDir(a6) ;make it current

    movea.l FirstFile,a0 ;find start of file list

***************** open destination file *****************

SaveLoop:
    move.l a0,FilePTR
    move.l a0,d1
```

```
   addi.l #SFName,d1          ;pointer to file name
   move.l #MODE_NEWFILE,d2
   jsr    _LVOOpen(a6)    ;open it
   move.l d0,FileHandle
   beq    SaveFailed

******************** write the file **********************

   movea.l FilePTR,a0
   move.l d0,d1           ;file handle
   move.l a0,d2
   addi.l #StoredFile,d2 ;pointer to data
   move.l SFLength(a0),d3 ;file length
   jsr    _LVOWrite(a6)
   cmpi.l #-1,d0          ;error?
   beq    SaveFailed1

   move.l FileHandle,d1
   jsr    _LVOClose(a6)  ;close the file

************************ next file ************************

   movea.l FilePTR,a0
   tst.l (a0)
   movea.l (a0),a0        ;link to next file
   bne    SaveLoop

SaveDone:
   move.l DestLock,d1
   jsr    _LVOUnLock(a6) ;free memory used for lock
```

```
    bsr   FreeAll          ;free memory used for files

    clr.l FirstFile        ;reinitialize list of files
    move.l #FirstFile,LastFile

    rts

********** restore original current directory ***********

RestoreDir:
   move.l OldDir,d1
   movea.l DosLibrary,a6
   jsr   _LVOCurrentDir(a6)
   rts

********** give back all the allocated memory ***********

FreeAll:
   movea.l FirstFile,a1
   tst.l FirstFile
   beq   FADone           ;any more to free?

   move.l (a1),FirstFile ;link forward
   move.l 4(a1),d0        ;length of file
   addi.l #38,d0         ; + 38 = length of block
   movea.l _AbsExecBase,a6
   jsr   _LVOFreeMem(a6) ;free this block

   bra   FreeAll

FADone:
   rts
```

```
** subroutine to copy name to buffer and null terminate *

BuffCopy:
   lea   Ø(a1,dØ.L),a2  ;find stopping address
DCLoop:
   move.b (aØ)+,(a1)+   ;copy dest spec to buffer
   cmpa.l a1,a2
   bne   DCLoop

   clr.b (a2)           null terminate
   rts

********* subroutine to perform INSTR function **********

* aØ.L: starting position
* d1.B: search for this character
* d2.L: length of string to search
* d1.L: position of first occurance of this character (offset from start)

InString:
   moveq.l #1,dØ        ;start at char #1
InStrLoop:
   cmp.b -1(aØ,dØ.L),d1
   beq   InStrDun

   addq.l #1,dØ
   cmp.l d2,dØ
   bls   InStrLoop

   clr.l dØ             ;char not found: return Ø
InStrDun:
   rts
```

```
   SECTION  data,DATA

DOS_Name:
   dc.b  'dos.library',Ø

BAMsg:
   dc.b  'Bad Arguments',13,1Ø
BALen EQU *-BAMsg

NoMemMSG:
   dc.b  'Not Enough RAM for copy',13,1Ø
NMMLen EQU *-NoMemMSG

FailedMSG:
   dc.b  'Unable to write file',13,1Ø
FailedLen EQU *-FailedMSG

NewLine:
   dc.b  13,1Ø

   SECTION  mem,BSS

   CNOP  Ø,4      ;long word align
DirItem:
   ds.b  fib_SIZEOF

DosLibrary ds.l 1 ;handle for DOS library

FirstFile ds.l 1  ;pointer to first StoredFile in list
LastFile ds.l 1   ;pointer to last StoredFile in list
FilePTR  ds.l 1   ;pointer to current StoredFile during save
```

```
FileHandle ds.l 1 ;handle of file being saved

DirLock  ds.l 1   ;lock for directory being examined
DestLock ds.l 1   ;lock for destination directory
OldDir   ds.l 1   ;lock for old directory
StdOut   ds.l 1   ;handle for default output device

CmdEnd   ds.l 1

DirLen   ds.l 1   ;length of path spec
DirPath:
   ds.b  80       ;buffer for directory name

FileLen  ds.l 1   ;length of file spec
FileSpec:
   ds.b  80       ;buffer for file spec

DestLen  ds.l 1   ;length of destination path
DestSpec:
   ds.b  80       ;buffer for destination path

FNameLen ds.l 1   ;length of filename

   END
```

Appendices

Appendix A

Amiga Character Codes

	Hex	Decimal	Keypress		Hex	Decimal	Keypress
	20	32	space bar	2	32	50	2
!	21	33	SHIFT-1	3	33	51	3
"	22	34	SHIFT-'	4	34	52	4
#	23	35	SHIFT-3	5	35	53	5
$	24	36	SHIFT-4	6	36	54	6
%	25	37	SHIFT-5	7	37	55	7
&	26	38	SHIFT-7	8	38	56	8
'	27	39	'	9	39	57	9
(	28	40	SHIFT-9	:	3A	58	SHIFT-;
)	29	41	SHIFT-0	;	3B	59	;
*	2A	42	SHIFT-8	<	3C	60	SHIFT-,
+	2B	43	SHIFT-=	=	3D	61	=
,	2C	44	,	>	3E	62	SHIFT-.
-	2D	45	-	?	3F	63	SHIFT-/
.	2E	46	.	@	40	64	SHIFT-2
/	2F	47	/	A	41	65	SHIFT-A
0	30	48	0	B	42	66	SHIFT-B
1	31	49	1	C	43	67	SHIFT-C

	Hex	Decimal	Keypress		Hex	Decimal	Keypress
D	44	68	SHIFT-D	X	58	88	SHIFT-X
E	45	69	SHIFT-E	Y	59	89	SHIFT-Y
F	46	70	SHIFT-F	Z	5A	90	SHIFT-Z
G	47	71	SHIFT-G	[	5B	91	[
H	48	72	SHIFT-H	\	5C	92	\
I	49	73	SHIFT-I	]	5D	93	]
J	4A	74	SHIFT-J	^	5E	94	SHIFT-6
K	4B	75	SHIFT-K	_	5F	95	SHIFT--
L	4C	76	SHIFT-L	`	60	96	`
M	4D	77	SHIFT-M	a	61	97	A
N	4E	78	SHIFT-N	b	62	98	B
O	4F	79	SHIFT-O	c	63	99	C
P	50	80	SHIFT-P	d	64	100	D
Q	51	81	SHIFT-Q	e	65	101	E
R	52	82	SHIFT-R	f	66	102	F
S	53	83	SHIFT-S	g	67	103	G
T	54	84	SHIFT-T	h	68	104	H
U	55	85	SHIFT-U	i	69	105	I
V	56	86	SHIFT-V	j	6A	106	J
W	57	87	SHIFT-W	k	6B	107	K

	Hex	Decimal	Keypress
l	6C	108	L
m	6D	109	M
n	6E	110	N
o	6F	111	O
p	70	112	P
q	71	113	Q
r	72	114	R
s	73	115	S
t	74	116	T
u	75	117	U
v	76	118	V
w	77	119	W
x	78	120	X
y	79	121	Y
z	7A	123	Z
{	7B	124	SHIFT-[
|	7C	125	SHIFT-\
}	7D	126	SHIFT-]
~	7E	127	SHIFT-'
	A0	160	ALT-space bar

	Hex	Decimal	Keypress
¡	A1	161	ALT-SHIFT-1
¢	A2	162	ALT-SHIFT-'
£	A3	163	ALT-SHIFT-3
¤	A4	164	ALT-SHIFT-4
¥	A5	165	ALT-SHIFT-5
¦	A6	166	ALT-SHIFT-7
§	A7	167	ALT-'
¨	A8	168	ALT-SHIFT-9
©	A9	169	ALT-SHIFT-0
ª	AA	170	ALT-SHIFT-8
«	AB	171	ALT-SHIFT-=
¬	AC	172	ALT-,
	AD	173	ALT--
®	AE	174	ALT-.
¯	AF	175	ALT-/
°	B0	176	ALT-0
±	B1	177	ALT-1
²	B2	178	ALT-2
³	B3	179	ALT-3
´	B4	180	ALT-4

	Hex	Decimal	Keypress		Hex	Decimal	Keypress
µ	B5	181	ALT-5	É	C9	201	ALT-SHIFT-I
¶	B6	182	ALT-6	Ê	CA	202	ALT-SHIFT-J
·	B7	183	ALT-7	Ë	CB	203	ALT-SHIFT-K
¸	B8	184	ALT-8	Ì	CC	204	ALT-SHIFT-L
¹	B9	185	ALT-9	Í	CD	205	ALT-SHIFT-M
º	BA	186	ALT-SHIFT-;	Î	CE	206	ALT-SHIFT-N
»	BB	187	ALT-;	Ï	CF	207	ALT-SHIFT-O
¼	BC	188	ALT-SHIFT-,	Ð	D0	208	ALT-SHIFT-P
½	BD	189	ALT-=	Ñ	D1	209	ALT-SHIFT-Q
¾	BE	190	ALT-SHIFT-.	Ò	D2	210	ALT-SHIFT-R
¿	BF	191	ALT-SHIFT-/	Ó	D3	211	ALT-SHIFT-S
À	C0	192	ALT-SHIFT-2	Ô	D4	212	ALT-SHIFT-T
Á	C1	193	ALT-SHIFT-A	Õ	D5	213	ALT-SHIFT-U
Â	C2	194	ALT-SHIFT-B	Ö	D6	214	ALT-SHIFT-V
Ã	C3	195	ALT-SHIFT-C	×	D7	215	ALT-SHIFT-W
Ä	C4	196	ALT-SHIFT-D	Ø	D8	216	ALT-SHIFT-X
Å	C5	197	ALT-SHIFT-E	Ù	D9	217	ALT-SHIFT-Y
Æ	C6	198	ALT-SHIFT-F	Ú	DA	218	ALT-SHIFT-Z
Ç	C7	199	ALT-SHIFT-G	Û	DB	219	ALT-[
È	C8	200	ALT-SHIFT-H	Ü	DC	220	ALT-\

	Hex	Decimal	Keypress		Hex	Decimal	Keypress
ý	DD	221	ALT-]	ï	EF	239	ALT-O
þ	DE	222	ALT-SHIFT-6	ð	F0	240	ALT-P
ß	DF	223	ALT-SHIFT--	ñ	F1	241	ALT-Q
à	E0	224	ALT-'	ò	F2	242	ALT-R
á	E1	225	ALT-A	ó	F3	243	ALT-S
â	E2	226	ALT-B	ô	F4	244	ALT-T
ã	E3	227	ALT-C	õ	F5	245	ALT-U
ä	E4	228	ALT-D	ö	F6	246	ALT-V
å	E5	229	ALT-E		F7	247	ALT-W
æ	E6	230	ALT-F	ø	F8	248	ALT-X
ç	E7	231	ALT-G	ù	F9	249	ALT-Y
è	E8	232	ALT-H	ú	FA	250	ALT-Z
é	E9	233	ALT-I	û	FB	251	ALT-SHIFT-[
ê	EA	234	ALT-J	ü	FC	252	ALT-SHIFT-\
ë	EB	235	ALT-K	ý	FD	253	ALT-SHIFT-]
ì	EC	236	ALT-L	þ	FE	254	ALT-SHIFT-'
í	ED	237	ALT-M	ÿ	FF	255	ALT-— (on numeric keypad)
î	EE	238	ALT-N				

Appendix B

AmigaDOS Command Summary

Sheldon Leemon and Arlan R. Levitan

This appendix is a command-by-command listing of AmigaDOS. For the most part, its format is self-explanatory. However, under the "Format" headings there are several typographical devices used to show you what is required and what is optional.

- Keywords which are required are in uppercase boldface roman type. **ASSIGN** and **COPY** are examples.
- Keywords which are optional are in uppercase boldface italics. ***LIST*** is an example.
- Optional entries are enclosed in brackets—[ ].
- Parameters are in lowercase italics. These denote where you'll enter something. If required, the parameter is not enclosed in brackets. If optional, it is enclosed in brackets. Thus,

  COPY [*FROM fromname*] [*TO toname*] [*ALL*] [*QUIET*]

  indicates that the keyword **COPY** is required, that the keywords ***FROM, TO, ALL,*** and ***QUIET*** are all optional, and that the two parameters *fromname* and *toname* are also optional.

ASSIGN

Builds, removes, and lists associations between logical device names and filing system directories, physical devices (DF1:, PRT:, and so on), and disk volume names.

Format

ASSIGN *devname dirname* [*LIST*]

devname The logical device name that you wish to assign to a directory, physical device, or disk volume.

dirname The directory path, physical device, or disk volume name that will be represented by references to the specified *devname.*

BREAK

Sets attention flags which interrupt a process as if the user had pressed specified CTRL-key combinations in an active window.

Format

BREAK *tasknum* [*C*] [*D*] [*E*] [*F*] [*ALL*]

tasknum The number assigned by the system to the CLI process that you wish to interrupt (for more information, see the STATUS command).

[*C*] [*D*] [*E*] [*F*] [*ALL*] The attention flag(s) associated with the interrupt type that you wish to issue.

CD

Sets or changes the current directory or drive. Also used to display the current drive and directory.

Format

CD [*name*]

name The name of the directory path or logical device name that you wish to make the current directory.

COPY

Copies one or more files or directories from one disk to another and as an option lets you give the copies a name different from the orginal(s).

COPY can also copy files to the same disk if different names are used for the copies or if they're copied to different directories.

Format

COPY [*FROM fromname*] [*TO toname*] [*ALL*] [*QUIET*]

[*FROM fromname*] Specifies the directory or file(s) you want copied. The keyword FROM is not needed as long as the files are named in the correct order (*fromfile*, then *tofile*). If you change the order (COPY TO *tofile* FROM *fromfile*), the keyword FROM is required.

[*TO toname*] Specifies the TO target (where you want to put the FROM files you are copying). The keyword TO is necessary only if the TO destination is listed *before* the FROM source.

[ALL] If you use this keyword, any files, subdirectories, and the files in the subdirectories located in *fromname*'s directory will be copied to the *toname* directory.

[QUIET] When copying multiple files (due to the use of pattern matching or the ALL keyword), the name of the files being copied and directories created are displayed unless this keyword is specified.

DATE

Used to display, change, or store the current setting of the system date and time. If you haven't bought and connected a separate clock/calendar accessory, AmigaDOS checks the boot-up disk for the date of the most recently modified or created file and sets the system date a bit in advance of that.

Format

DATE [*date*] [*time*] [*TO* or *VER name*]

date The day of the month, the month, and the year to which the system date will be set. A specific desired date is typed in as *DD-MM-YY.*

time The time of day to which the system clock is to be set. The time should be entered in the form *HH:MM:SS,* representing hours, minutes, and seconds of the desired clock setting.

[*TO* or *VER name*] The TO and VER options allow you to store the present system date and time to *name,* which may be a disk file or a physical device such as a printer.

DELETE

Removes files and directories from the designated drive. If no drive is designated, the current default drive is assumed. If no directory path is specified, the files and/or directories are deleted from the current directory. DELETE accepts patterns as well as specific filenames.

Format

DELETE *name,,,,,,,,*[*ALL*] [*Q* or *QUIET*]

name The name of the file(s) or directory entry(s) to be removed. Up to ten file or directory names may be entered within a single DELETE command.

[*ALL*] When this keyword is used, DELETE erases all files and subdirectories contained within the directory as well as the directory itself.

[*QUIET*] Suppresses the status reports that are issued as each file's deletion is attempted during a DELETE which erases more than one file.

DIR

Lists the file and subdirectories with the present directory or another specified directory. The list is normally grouped into a list of subdirectories, followed by a sorted list of files. Options available for use with this command allow you to use a special interactive mode and/or ask for an extended listing which lists the contents of subdirectories as well.

Format

DIR *dirname* [*OPT A* or *OPT I* or *OPT AI*]

dirname The name of the directory or logical device whose contents you want displayed. An AmigaDOS pattern may also be used to display multiple directories. If no directory or AmigaDOS pattern is specified, the current directory is displayed.

[*OPT A* or *OPT I* or *OPT AI*] When the OPT A keyword is used, the display includes the contents of any subdirectories residing in the directory being listed. This lets you see everything in a directory with a single command.

OPT I invokes the special interactive mode of DIR. In interactive mode your system pauses as each subdirectory entry or file is listed, displaying a question mark to the right of the entry. When in interactive mode, you may use any of the following subcommands:

Key(s)	Function
<RETURN>	Doesn't do anything with the current item. Goes on to the next item in the DIR listing.
T <RETURN>	Types (lists) the file. To pause the display while listing, hit the space bar or any key. To resume after pausing, press the BACK SPACE key or CTRL-X. When you want to abandon the listing of the file contents before the complete file has been listed, type CTRL-C. You'll be returned to the interactive mode. T is an invalid option for subdirectories.

DEL <RETURN> Erases the file. Subdirectories may be erased only if they're empty.

E <RETURN> Enters a subdirectory. Displays the files and subdirectories within a subdirectory. The listing remains in interactive mode. Not a valid option for a file.

B <ENTER> Goes back to the previous DIR item, still in interactive mode. This lets you back up in case you pass by an item you later decide you want to act on.

Q <ENTER> Quit. Abandons the DIR listing and goes back to the CLI prompt.

OPT AI combines both the A and I options, resulting in an interactive listing of all files and directories within the specified directory.

DISKCOPY

Makes duplicates of the entire contents of 3½-inch disks. When you use DISKCOPY, any information previously stored on the destination disk is erased.

Format

DISKCOPY [*FROM*] *source drive* **TO** *destination drive* [*NAME volname*]

[*FROM*] *source drive* The name of the drive in which the disk you wish to copy will be mounted.

TO *destination drive* The TO keyword *must* be used with the DISKCOPY command. This is the name of the drive in which the disk to be copied to will be mounted.

[*NAME volname*] The volume name that will be given to the copy of the original disk.

ECHO

ECHO is used in command files to display a message on the system screen. This is most often helpful when the RUN command is being used to carry out a background operation whose completion would otherwise not be readily apparent to the user.

Format

ECHO *string*

string The message to be written to the currently active output stream. If it contains spaces, *string* should be contained within quotation marks.

ED

The ED command is used to edit the contents of a file using AmigaDOS's full-screen editor.

Format

ED [*FROM*] *name* [*SIZE*] *n*

[*FROM*] *name* The name of the AmigaDOS file which you wish to edit using the full-screen editor. If *name* is the first argument in an ED command statement, the FROM keyword need not be specified.

[*SIZE*] *n* Used to set the size of the editor's workspace. If *n* is the second argument in an ED command statement, the SIZE keyword need not be given.

EDIT

The EDIT command is used to edit the contents of a file using AmigaDOS's line editor.

Format

EDIT [*FROM*] *fromname* [*TO*] *toname* [*WITH*] *withname* [*VER*] *vername* [*OPT option*]

[*FROM*] *fromname* The name of the file whose contents will be edited. If *fromname* is the first argument in the EDIT command, the FROM keyword is optional. EDIT requires *fromname,* and it must already exist.

[*TO*] *toname* The name of the file to which the edited text is saved when a Q or W subcommand is executed from within the line editor. If *toname* is the second argument in an EDIT command (following *fromname*), the TO keyword is optional.

[*WITH*] *withname* Lets you specify a file which will be used as input to the line editor's command processor. The contents of *withname* should be a series of valid line editor subcommands. If *withname* is the third argument in an EDIT command (following *fromname* and *toname*), the WITH keyword is optional.

[VER] vername Lets you specify where you want messages and verification output produced by the line editor sent; *vername* may be a file or logical device. If *vername* is the fourth argument in an EDIT command (following *fromname, toname,* and *withname*), the VER keyword is optional.

[OPT Pn or *OPT Wn* or *OPT PnWn]* These options let you set the maximum line length (W*n*) and number of lines (P*n*) that EDIT will keep memory resident. The default maximum line length is 120. The default number of lines is 40. Multiplying the value for P*n* by W*n* yields the amount of memory that EDIT reserves as a temporary work area. If either P*n* or W*n* is to be specified, the OPT keyword must be used.

ENDCLI

ENDCLI terminates the current Command Line Interpreter.

Format

ENDCLI

EXECUTE

The EXECUTE command is used to invoke AmigaDOS command sequence files. Command sequence files contain a prestored series of commands which are executed sequentially once the command file has been started by EXECUTE.

Format

EXECUTE *name [arg1 arg2,,,,]*

name The name of the command sequence file to be invoked; *name* is a required parameter and may be any valid AmigaDOS filename.

[arg1 arg2,,,,] Arguments to be passed to the command sequence file. Arguments may be any valid AmigaDOS string (including filenames and logical and physical devices).

FAILAT

The FAILAT command is used within command sequence files and RUN command statements to alter the failure level threshold of the system.

When AmigaDOS commands encounter an error upon execution, a numeric return code is set (usually 5, 10, or 20). The higher the return code, the greater the severity of the error. If a return code which exceeds the current failure level

threshold is encountered during execution of a command sequence file or multiple command task set up by a RUN, execution stops. The default failure level threshold of AmigaDOS command sequence files and RUN background tasks is 10.

Format

FAILAT *n*

n The new failure level threshold. If *n* is not specified, FAILAT displays the current failure level threshold.

FAULT

The FAULT command provides English-language explanations for many of the error codes which AmigaDOS generates.

Format

FAULT *n,,,,,,,,*

n,,,,,,,, The error number (fault code) which you want explained. Up to ten error numbers may be specified within one FAULT command. If no information is available on the error, the system simply repeats the error number.

FILENOTE

FILENOTE lets you store comments about AmigaDOS files. Any comments stored using FILENOTE remain distinct and separate from the actual contents of the file. Comments stored using FILENOTE may be viewed by using the LIST command.

Format

FILENOTE [*FILE*] *filename* [*COMMENT*] *string*

[*FILE*] *filename* The name of the file that is to have a comment attached.

[*COMMENT*] *string* Defines the comment assigned to the specified file. The COMMENT keyword is optional if *string* is the second argument of a FILENOTE statement (following *filename*); *string*, the comment to be attached to the file, can be up to 80 characters in length and must be enclosed in quotation marks if it contains spaces.

FORMAT

Initializes a floppy disk as a blank AmigaDOS disk. *Caution: If a used disk is formatted, all information on it will be erased.*

Format

FORMAT DRIVE *drivename* **NAME** *string*

DRIVE *drivename* The disk drive in which you will insert the disk that's to be formatted. The DRIVE keyword *must* be used. The valid values for *drivename* are df0:, df1:, df2:, and df3:.

NAME *string* The volume name assigned to the formatted disk. The NAME keyword is mandatory. *String* is the name you want to call the disk, and it must also be specified; *string* can be up to 30 characters long and must be enclosed in quotation marks if it contains spaces.

IF-ELSE-ENDIF

The IF command and its associates (the ELSE and ENDIF commands) are used within AmigaDOS command sequence files to carry out groups of commands within the command sequence file *if* one or more conditions are met. If an IF statement is satisfied, the commands following the statement are executed sequentially until an ELSE or ENDIF statement is encountered. If the IF conditional is not satisfied and an ELSE statement is encountered before an ENDIF, the commands between ELSE and ENDIF are executed.

Format

IF [*NOT*] [*WARN*] [*ERROR*] [*FAIL*] [<*string1*> *EQ* <*string2*>] [*EXISTS name*]

[*NOT*] Reverses the result of the IF test. If any of the conditionals is true and NOT is also used, the IF statement will not be satisfied. If all the other specified conditionals are false and NOT is used, the IF statement will be satisfied.

[*WARN*] Is satisfied (true) if the return code of the previous command is greater than or equal to 5.

[*ERROR*] Is satisfied (true) if the return code of the previous command is greater than or equal to 10.

[*FAIL*] Is satisfied (true) if the return code of the previous command is greater than or equal to 20.

[<*string1*> *EQ* <*string2*>] Is satisfied (true) if *string1* is identical to *string2*. Case is ignored.

[*EXISTS name*] Is satisfied if *name* exists; *name* may be any AmigaDOS file or directory.

INFO

Displays information about disk volumes and the system RAM disk.

Format

INFO

INSTALL

The INSTALL command makes a formatted disk capable of a minimal startup of the AmigaDOS environment (assigning SYS: to the booted disk).

Format

INSTALL [*DRIVE*] *drive*

[*DRIVE*] *drive* The disk drive in which the disk you wish to make bootable resides. The DRIVE keyword is optional. Valid values for *drive* are df0:, df1:, df2:, and df3:.

JOIN

JOIN lets you merge the contents of up to 15 files into one file. The files are merged in the order given to JOIN.

Format

JOIN *name1 name2 ,,,,,,,,,,,,* **AS** *destname*

name1 name2 ,,,,,,,,,,,, The names of the files you want merged together. A minimum of 2 files must be given, with a space between each name. Up to 15 files may be merged by a single JOIN command.

AS *destname* The name of the file that the contents of all files preceding the AS keyword (which is required) will be merged into; *destname* can be a new or old file, but it cannot be any of the files which precede the AS keyword. If *destname* already exists, its previous contents will be replaced.

LAB

LAB is used within command sequence files to define a location in the command file that may be jumped to by the SKIP command.

Format

LAB *string*

string A "signpost" that can be used by a SKIP command to jump to the spot in the command file where a specific LAB statement is located. Once jumped to, command file execution continues with the commands following the LAB statement.

LIST

Displays the name, size, protection status, time and date of creation, and the Amiga filing system block numbers of *(a)* a directory, *(b)* a selected portion of a directory, or *(c)* a single file. LIST also displays any comments attached to a file by a FILENOTE command.

Format

LIST *listname* [*P* or *PAT pattern*] [*KEYS*] [*DATES*] [*NODATES*] [*TO device* or *filename*] [*S string*] [*SINCE date*] [*UPTO date*] [*QUICK*]

listname Can be the device name or volume name of a disk, a directory, or the name of a specific file.

[*P* or *PAT pattern*] When you use this option, the P or PAT keyword must precede the pattern. A pattern allows you to specify a number of files, each of which has some common characteristic.

[*KEYS*] Specifying this option includes the block number associated with each file and directory displayed.

[*DATES*] Includes file and directory creation date and time information in the LIST display.

[*NODATES*] Instructs LIST to suppress the display of file and directory creation date and time information.

[*TO device* or *filename*] Selects where the output of LIST is to be sent; *device* or *filename* may be any valid AmigaDOS filename or a logical device known to the system. If a file of the same name already exists, the existing file will be deleted and a new file with the same name is created.

[*S string*] To use this option, the S keyword must precede *string*, which can be any character string. LIST then displays only those filenames or directories which include *string*. If spaces are included in *string*, quotation marks must enclose it.

[*SINCE date*] Displays information only for those files and directories created or modified on or after *date; date* may be specified in the format *DD-MMM-YY,* or as an indirect reference of YESTERDAY, TODAY, or TOMORROW. The days of the past week, SUNDAY through SATURDAY, can also be used as *date.*

[*UPTO date*] Instructs LIST to display information only for those files and directories created or modified on or before *date,* which is subject to the same restrictions as the SINCE keyword.

[*QUICK*] Instructs LIST to display only file and directory names. However, if the DATES and/or KEYS keywords are specified as well, LIST displays file and directory names along with the information associated with DATES and/or KEYS.

MAKEDIR

MAKEDIR creates directory entries, allowing you to partition an AmigaDOS disk into a type of multileveled filing cabinet.

Format

MAKEDIR *name*

name The *name* of the directory to be created; *name* must be specified. MAKEDIR fails if *name* is the name of a file or subdirectory which already exists in the "parent" directory (the next highest directory in the hierarchy). MAKEDIR also fails if a nonexistent pathname is specified.

NEWCLI

NEWCLI opens a new CLI window on the system display.

Format

NEWCLI [*CON: hpos/vpos/width/height/windowtitle*]

[*CON: hpos/vpos/width/height/windowtitle*] CON: lets you specify the size, position, and title of the new CLI window. CON: is required if *any* of the following parameters is specified.

- *hpos* is the horizontal position of the top left corner of the window (expressed as the number of pixels in from the left edge of the screen). If a value for *hpos* is omitted, it's assumed to be zero.

- *vpos* is the vertical position of the top left corner of the window (expressed as the number of pixels down from the top edge of the screen). If a value for *vpos* is omitted, it's assumed to be zero.
- *width* and *height*, which must be specified, give the size of the window in pixels. The maximum size for a CLI window is the screen size, 640 × 200 pixels. The minimum is 90 × 25 pixels.
- *windowtitle*, which is optional, allows you to enter the text of a title to appear in the title bar. If you want to set *windowtitle*, all preceding parameters must also be set.

PROMPT

The PROMPT command changes the CLI prompt for the currently active CLI. The default prompt for any given CLI is *n*>, where *n* is the task number associated with that CLI.

Format

PROMPT *prompt*

prompt The string you want to substitute for the active CLI's prompt. If no value for *prompt* is specified, the CLI prompt will be changed to >; *prompt* may be a maximum of 59 characters. If it contains spaces, the entire prompt must be enclosed by double quotation marks.

PROTECT

PROTECT allows you to alter the attributes of AmigaDOS files and directory entries. There are protection flags associated with each of four attributes. The flags are *r, w, e,* and *d;* they tell the system if the file or directory entry may be read (r), written over (w), executed (e), or deleted (d).

It is important to note that in the initial releases of AmigaDOS (1.0 and 1.1), only the Delete flag works. You can set the others, but DOS does not act on those settings.

Format

PROTECT [*FILE*] *name* [*FLAGS*] [*R*][*W*][*E*][*D*]

[*FILE*] *name* The name of the file whose protection flags are to be modified; *name,* which is mandatory, may be any valid AmigaDOS filename or directory name. The FILE keyword is optional.

[FLAGS] *[R][W][E][D]* The protection flags which will be turned on by PROTECT. The FLAGS keyword does not have to be entered—it's optional. The protection flags to be turned on must be specified as a single string in any desired order. Remember that if a flag is set to on, the operation associated with the flag may be carried out. If no flags are specified, all flags are turned off. These are the operations associated with each flag:

R Read
W Write
E Execute
D Delete

QUIT

The QUIT command is used within command sequence files to exit the command sequence file and, optionally, to set the return code.

Format

QUIT [*returncode*]

returncode The return code which is reported when the command sequence file is terminated by a QUIT. If *returncode* is nonzero, the message

quit failed returncode *returncode*

is displayed on the screen, with the number specified substituted for *returncode*. If *returncode* is set to zero or is not specified, no message is displayed on termination of the command sequence file by QUIT.

RELABEL

RELABEL lets you change the volume name associated with a floppy disk.

Note: RELABEL does *not* prompt you for the disk to be inserted. If you have a single-drive system and insert the disk you wish to relabel ahead of time and then issue the RELABEL command, you'll be prompted to insert the disk with the command library on it in any disk drive. Once you do so, RELABEL promptly renames the volume with the command library on it. The following procedure will work for single-drive system owners.

```
COPY :C/RELABEL TO RAM:
RUN RAM:RELABEL df0: NewName
```

Format

RELABEL [***DRIVE***] *drive* [***NAME***] *name*

[***DRIVE***] *drive* The disk drive in which the disk to be relabeled is mounted. The DRIVE keyword is optional if *drive* precedes the volume name in the RELABEL statement.

[***NAME***] *name* The volume name which will replace whatever name is currently associated with the target disk; *name* may be up to 30 characters long. If the volume name contains spaces, quotation marks must enclose it. The NAME keyword is optional if *name* follows *drive.*

Note: Under AmigaDOS version 1.0, RELABEL fails if no drive is specified or if *name* is omitted.

RENAME

RENAME allows you to change the name of AmigaDOS files and directories. AmigaDOS's RENAME function also lets you move files from one directory to another on the same disk and reorganize directory structures at will.

Format

RENAME [***FROM***] *fromname* [***TO*** or ***AS***] *toname*

[***FROM***] *fromname* The file or directory that's to be renamed. The FROM keyword is not required if *fromname* is the first argument of a RENAME statement.

[***TO*** or ***AS***] *toname* The new name to be given to the file or directory specified by *fromname.* The TO and AS keywords may be used interchangeably and are optional if *toname* is the second argument of a RENAME statement. If *fromname* already exists, RENAME will fail.

Note: fromname and *toname* must reside on the same disk volume.

RENAME's ability to manipulate AmigaDOS directory structures makes this one of the most powerful AmigaDOS commands and, consequently, a command that should be used with great care. An entire directory, including all files, subdirectories, and files within its subdirectories may be moved to another location in the volume's directory tree structure with a single RENAME.

RUN

The RUN command may be used to create a system CLI task which executes in the Amiga's background (in other words, the task doesn't present you with an interactive CLI window). RUN allows multiple AmigaDOS commands to be executed in sequence. Once all commands given to a RUN statement are executed, the background task disappears.

Format

RUN *command+command,,,,,,,*

command+command,,,,,,, The AmigaDOS command you want executed in the background. More than one command may be strung together in a RUN sequence.

SAY

The SAY command is used to invoke the Amiga's built-in speech synthesis capabilities. The quality and speed of speech may be controlled by the user. SAY has two modes—interactive and direct.

In direct mode, the text to be spoken or an AmigaDOS file containing the text to be spoken is specified on the command line with the keyword SAY.

Interactive mode is entered by typing SAY by itself. Two windows will appear on the system screen.

The *Phoneme window* initially displays the option codes which may be used to control the quality and speed of the synthesized voice. As text is spoken, the phoneme codes that SAY uses are displayed.

The *Input window* is where text you wish spoken is displayed as it's typed in. The text is passed to SAY when the RETURN key is pressed. The interactive mode is exited by typing a line consisting only of a RETURN keystroke.

The SAY command was added to AmigaDOS in release 1.1.

Format

SAY [*options*] [*text*],,,,,,,,,

[*options*] Controls the quality, pitch, speed, and source of the text to be spoken. SAY identifies an option by a leading dash (–). These are valid options for SAY:

Option	Function
-f	Use female voice.
-m	Use male voice.
-n	Use natural voice.
-r	Use robot voice (monotone).
-p###	Set pitch of voice to ### (valid values are 65–320).
-s###	Set speech rate to ### (valid values are 40–400).
-x *file*	Say contents of *file*. The -x option may not be invoked in the interactive mode of SAY; *file* must be an AmigaDOS file in the current directory and may not contain *any* spaces or be enclosed in parentheses.

Multiple options, separated by spaces, may be specified at one time.

[*text*] The text to be spoken.

SEARCH

SEARCH lets you scan AmigaDOS files for a specified string of characters. You may SEARCH a single file, all files matching an AmigaDOS pattern, all files within a directory, and, optionally, all files within a directory's subdirectories.

Format

SEARCH [*FROM*] *name* [*SEARCH*] *string* [*ALL*]

[*FROM*] *name* The file or directory that you want searched; *name* may also be an AmigaDOS pattern. If *name* is the first argument in the SEARCH command, the FROM keyword is optional.

[*SEARCH*] *string* The text string that will be searched for. If *string* is the second argument in the SEARCH command, this second SEARCH keyword is optional. If *string* contains any spaces, it must be enclosed in quotation marks.

[*ALL*] If the ALL keyword is specified and *name* is an AmigaDOS directory, all files within the directory and its subdirectories are searched.

; (Semicolon)

The semicolon (;) command allows the insertion of informational comments in command sequence files.

Format

; [*comment*]

[*comment*] May be any text string, up to 254 characters in length (if the ; is the first character of a line).

SKIP

The SKIP command is used within command sequence files to jump to a specified label. If a SKIP is executed, command execution continues immediately after the label which was skipped to.

Format

SKIP [*string*]

[*string*] The string attached to a LAB command which SKIP searches for in the currently executing command file. The search starts at the command following SKIP and continues downward toward the end of the command file. If the matching LAB *string* command precedes the SKIP command, SKIP will not find it, and the command file terminates with an error.

If *string* is not specified, the first LAB command following SKIP will be skipped to.

SORT

SORT performs an alphabetic sort on contents of an AmigaDOS text file. SORT is line-oriented.

Format

SORT [*FROM*] *fromname* [*TO*] *toname* [*COLSTART n*]

[*FROM*] *fromname* The name of the AmigaDOS file whose contents are to be sorted. If *fromname* is the first argument of a SORT command, the FROM keyword is optional.

[*TO*] *toname* The name of the AmigaDOS file or logical device that the sorted lines from *fromname* will be sent to. If *toname* is the second argument of a SORT command, the TO keyword is optional; *toname* must be different from *fromname* or the SORT will fail.

[*COLSTART n*] Lets you specify that SORT will compare lines beginning with the *n*th character in each line. If *n* is given, the COLSTART keyword *must* be used. If COLSTART *n* has been specified and lines are found to be equal, SORT attempts a secondary sort of the equal lines, starting with the first character of each line.

STACK

The STACK command may be used to display or set aside the amount of stack space for the currently active CLI.

Format

STACK [*n*]

[*n*] The amount of space, in bytes of memory, that you wish to assign as stack space for the currently active CLI. If *n* is omitted, the current stack size is displayed.

STATUS

The STATUS command displays system information about active tasks.

Format

STATUS [*tasknum*] [*FULL*] [*TCB*] [*SEGS*] [*CLI* or *ALL*]

tasknum The number of the task which STATUS is to report on. If *tasknum* is not specified, all active tasks are reported.

[*FULL*] Displays all the information normally reported by STATUS if the TCB, SEGS, and ALL keywords were all specified. The FULL keyword is optional.

[*TCB*] Causes STATUS to display information dealing with the stack size, global vector size, and priority of each active task known to the system. The TCB keyword is optional.

[*SEGS*] Causes STATUS to display each active task's segment list section names. The SEGS keyword is optional.

[*CLI* or *ALL*] Specifying either CLI or ALL causes STATUS to report on all currently active CLI tasks and display the section names of all commands currently loaded within the CLI. The CLI and ALL keywords are interchangeable and optional.

TYPE

The TYPE command lets you output the contents of any AmigaDOS file to the screen, a disk file, or any AmigaDOS physical device.

Format

TYPE [*FROM*] *fromname* [[*TO*] *toname*] [*OPT N* or *OPT H*]

[*FROM*] *fromname* The name of the file you want TYPEd; *fromname* is required and may be any valid AmigaDOS filename. The FROM keyword is optional and need not be specified if *fromname* immediately follows TYPE.

[[*TO*] *toname*] The name of the file or device you want the output of the TYPE operation sent to. The TO keyword is

optional if the first argument of TYPE is *fromname* and the second argument is *toname*. If no destination for TYPE's output is specified, the output is displayed on the screen.

[***OPT N*** or ***OPT H***] Adding *OPT N* to a TYPE command instructs the system to precede each line output by TYPE with a line number. AmigaDOS treats any number of characters within a file ending with a linefeed as one line.

Specifying *OPT H* instructs TYPE to produce a formatted hexadecimal dump of the *fromname* file's contents. The *N* and *H* options are mutually exclusive—only one may be specified. If either option is desired, the OPT keyword *must* be used.

WAIT

WAIT can be used to put a task in a state of suspended animation for a user-definable period of time or until a specified time of day. WAIT can be used in command sequence files or in conjunction with a RUN command.

Format

WAIT [*n*] [***SEC*** or ***SECS***] [***MIN*** or ***MINS***] [***UNTIL*** *time*]

[*n*] [***SEC*** or ***SECS***] [***MIN*** or ***MINS***] The amount of time, in minutes or seconds, that the system will wait. If *n* is omitted and the SEC or MIN keyword is specified, *n* defaults to 1. Using the SEC or SECS keyword tells AmigaDOS to wait *n* seconds, while using MIN or MINS causes the CLI task to wait *n* minutes before continuing. SEC/SECS and MIN/MINS keywords are optional. If they're omitted, the default unit of time is seconds.

[***UNTIL*** *time*] The time of day you want the current process to wait until before continuing. If *time* is specified, the UNTIL keyword is required; *time* must be stated in the format *HH:MM*, where *HH* and *MM* are the hour and minute of the day in military (24-hour) time. If UNTIL *time* is used, the system will "wake up" sometime between *HH:MM*:00 and *HH:MM*:59.

WHY

WHY can be used to obtain additional information about failing commands.

Format

WHY

Appendix C

Frequency Values for Equal-Tempered Musical Scale

The following frequency numbers approximate the 88 notes on a piano. These values can be used as frequency parameters for SOUND commands in AmigaBASIC or as the basis for a pitch table in machine language programs. Each number has been rounded off to three decimal places. You can obtain more precise values by generating a frequency table mathematically with an AmigaBASIC routine similar to the one illustrated in Chapter 6.

Note	Octave	Pitch	Frequency Number
1	0	A	27.500
2	0	A#	29.135
3	0	B	30.868
4	0	C	32.703
5	0	C#	34.648
6	0	D	36.708
7	0	D#	38.891
8	0	E	41.203
9	0	F	43.654
10	0	F#	46.249
11	0	G	48.999
12	0	G#	51.913
13	1	A	55.000
14	1	A#	58.270
15	1	B	61.735
16	1	C	65.406
17	1	C#	69.296
18	1	D	73.416
19	1	D#	77.782
20	1	E	82.407
21	1	F	87.307
22	1	F#	92.499
23	1	G	97.999
24	1	G#	103.826

Note	Octave	Pitch	Frequency Number
25	2	A	110.000
26	2	A#	116.541
27	2	B	123.471
28	2	C	130.813
29	2	C#	138.591
30	2	D	146.832
31	2	D#	155.563
32	2	E	164.814
33	2	F	174.614
34	2	F#	184.997
35	2	G	195.998
36	2	G#	207.652
37	3	A	220.000
38	3	A#	233.082
39	3	B	246.942
40	3	C	261.626
41	3	C#	277.183
42	3	D	293.665
43	3	D#	311.127
44	3	E	329.628
45	3	F	349.228
46	3	F#	369.994
47	3	G	391.995
48	3	G#	415.305
49	4	A	440.000
50	4	A#	466.164
51	4	B	493.883
52	4	C	523.251
53	4	C#	554.365
54	4	D	587.330
55	4	D#	622.254
56	4	E	659.255
57	4	F	698.456
58	4	F#	739.989
59	4	G	783.991
60	4	G#	830.609

Note	Octave	Pitch	Frequency Number
61	5	A	880.000
62	5	A#	932.328
63	5	B	987.767
64	5	C	1046.502
65	5	C#	1108.731
66	5	D	1174.659
67	5	D#	1244.508
68	5	E	1318.510
69	5	F	1396.913
70	5	F#	1479.978
71	5	G	1567.982
72	5	G#	1661.219
73	6	A	1760.000
74	6	A#	1864.655
75	6	B	1975.533
76	6	C	2093.005
77	6	C#	2217.461
78	6	D	2349.318
79	6	D#	2489.016
80	6	E	2637.020
81	6	F	2793.826
82	6	F#	2959.955
83	6	G	3135.963
84	6	G#	3322.438
85	7	A	3520.000
86	7	A#	3729.310
87	7	B	3951.066
88	7	C	4186.009

Appendix D

Lattice C Compiler Flags

The Lattice C compiler is in two passes, LC1 and LC2, which have the following options.

LC1

-b Forces all static and external data to be addressing using a base register (a5 or a6). This limits the size of static data objects to 64K. It must be used if you want to produce position-independent code.

-c Precedes the following compatibility control flags. Any number of these flags may follow the -c, but there must be no spaces.

c—Allows comments to be nested. The current default does not allow nesting of comments.

d—Allows the dollar sign ($) to be used in identifiers.

m—Allows the use of multiple character constants.

s—Forces the compiler to produce only one copy of identical string constants. The current default is to produce unique copies of each string constant.

u—Causes all char declarations to be interpreted as unsigned char.

w—Turns off warnings generated by return statements that do not specify a return value from within functions declared as ints.

-d Forces debugging information to be included in the quad file generated by the first pass compiler.

-dSYMBOL Allows the identifier SYMBOL to be defined during compile. To set a value to the symbol, use -dSYMBOL=something.

-iPREFIX Tells the compiler where to find the include files. You may have up to four of these. This PREFIX is appended to the beginning of all include file requests.

-l Makes the compiler align data to addresses evenly divisible by four.

-n Causes the compiler to use only the first 8 characters of the identifiers to distinguish them. Generally the compiler uses the first 31 characters.

-oNAME Changes the name of the output file to NAME.Q.

-p Makes the compiler produce preprocessor output. The output will be in the file with the extension .p. The .q file will not be produced.

-u Cancels all automatic symbol definitions for this compilation.

-x Forces the compiler to interpret all external declarations as external reference rather than external definitions. In other words, the compiler normally allocates storage for an external variable and makes it publicly accessible. This option tells the compiler to assume that some other object module has already allocated the space and that these externs are references to them.

LC2

-f*n* Specifies which address variable should be used as the stack frame pointer—a5 or a6; *n* is either 5 or 6. If -b is used in LC1, whichever register is not used for the stack frame pointer will be used for the addressing of static and external data structures.

-oNAME Changes the name of the output file to NAME.o.

-r Forces all function calls to be made relative to the PC. This forces the range of function calls to be ±32K. This must be used if position-independent code is desired.

-s Adds a section name to this object module. ALink will merge all modules with the same name into one load unit.

-v This flag tells the compiler not to include stack checking in the code for entering functions. This has several effects: First, it improves the performance of your C program by around 20 percent (depending on the number of function calls your program makes). It also reduces the size of your code 14 bytes per function (since the stack-checking code is eliminated). However, it also means the runtime package may not detect a stack error. Once you have debugged a program, it's probably a good idea to use this flag.

Appendix E

Selected Intuition Routines

AddGadget( )

Adds a gadget to the gadget list of a window or screen.

SHORT AddGadget(Pointer, Gadget, Position);

Registers a0—Pointer; a1—Gadget; d0—Position

Pointer Pointer to the Window or Screen structure to which the gadget is to be added.

Gadget Pointer to the Gadget structure which is being added to the gadget list.

Position Integer position in the list for the new gadget. If this is zero, the gadget is added to the head of the list. If position is one, the gadget is inserted between the first and second gadget, and so on. Using a position greater than the number of gadgets in the list inserts the new gadget at the end of the list.

See also RemoveGadget( ).

AllocRemember( )

Allocates a block of memory and a Remember structure, and links the Remember structure into the list of allocated blocks.

APTR AllocRemember(RememberKey, Size, Flags);

Registers a0—RememberKey; d0—Size; d1—Flags

RememberKey An address to a pointer to a Remember structure. Before the first call to AllocRemember, the RememberKey should be set to NULL.

Size The number of bytes to allocate.

Flags The specifications for the memory to be allocated. These are the same as the EXEC function AllocMem( ); please refer to AllocMem documentation in the *ROM Kernel Manual* for more details.

Note: AllocRemember( ) calls the EXEC function AllocMem( ). AllocRemember( ) will return a NULL if the memory could not be allocated. Otherwise, it returns a pointer to the block of memory that was allocated.

See also FreeRemember( ); AllocMem( ) in Appendix F.

ClearMenuStrip( )

Clears the menu strip from a window.

ClearMenuStrip(Window);

Registers a0—Window

Window The pointer to the window which has the menu you are removing.

See also SetMenuStrip( ).

CloseScreen( )

Closes a screen.

CloseScreen(Screen);

Registers a0—Screen

Screen A pointer to the Screen structure of the screen you want to close.

Note: Everything which is rendered in the screen must be shut down before the screen is closed.

See also OpenScreen( ).

CloseWindow( )

Closes a window.

CloseWindow(Window);

Registers a0—Window

Window A pointer to the Window structure you want to close.

Note: If this window has menus, then you must ClearMenuStrip( ) on the window before closing the window. In addition, the menu cannot be displayed on the screen when CloseWindow( ) is called.

See also OpenWindow( ).

FreeRemember( )

Frees memory that was allocated using AllocRemember( ) to the system.

FreeRemember(RememberKey, Flag);

Registers a0—RememberKey; d0—Flag

RememberKey Address of the pointer to the Remember structure used with AllocRemember( ).

Flag If a Boolean True, free up both the Remember structures and the associated allocated memory blocks. If False, free up only the Remember structures and leave the allocated memory blocks intact.

See also AllocRemember( ).

ItemAddress( )

Returns the address of the specified MenuItem.

struct MenuItem *ItemAddress(MenuStrip, MenuNumber);

Registers	a0—MenuStrip; d0—MenuNumber
MenuStrip	A pointer to the first Menu structure in the menu strip holding the MenuItem you are trying to find.
MenuNumber	The value returned in a MENUPICK IntuiMessage. If this is MENUNULL, ItemAddress returns NULL.

ModifyIDCMP( )

Changes the state of a window's IDCMP flags.

ModifyIDCMP(Window, NewIDCMPFlags);

Registers	a0—Window; d0—NewIDCMPFlags
Window	A pointer to the Window structure holding the IDCMP you want to change.
NewIDCMPFlags	The new flag bits which you want to use.

Note: If you open a window with IDCMPFlags = NULL, the window will be opened with no IDCMP. If you call ModifyIDCMP( ) with flags on that window, the ports will be opened for you. The opposite if also true; if you start with IDCMP flags and call ModifyIDCMP( ) with NewIDCMPFlags = NULL, then the ports will be closed.

ModifyProp( )

Changes the parameters of a proportional gadget.

ModifyProp(Gadget, Pointer, Requester, Flags, HorizPot, VertPot, HorizBody, VertBody);

Registers	a0—Gadget; a1—Pointer; a2—Requester; d0—Flags; d1—HorizPot; d2—VertPot; d3—HorizBody; d4—VertBody
Gadget	The pointer to the Gadget structure we are going to modify.
Pointer	A pointer to the graphics element the gadget appears in (in general, this would be a pointer to a Window structure).
Requester	Points to a Requester structure if this is a requester gadget. Otherwise, set this to NULL.
Flags, HorizPot, VertPot, HorizBody, VertBody	The same as in the PropInfo structure.

OffGadget( )

Disables a gadget.

OffGadget(Gadget, Pointer, Requester);

Registers a0—Gadget; a1—Pointer; a2—Requester

Gadget Points to the Gadget structure which represents the gadget we are disabling.

Pointer A pointer to the graphics element the gadget appears in (a pointer to a Window structure if this gadget appears in a window).

Requester Points to a Requester structure if this is a requester gadget. Otherwise, set this to NULL.

Note: When a gadget is disabled, the gadget is ghosted, the GADGDISABLED flag is set, and the user is no longer allowed to select the gadget. If you are modifying a requester gadget, the requester must be displayed.

See also OnGadget( ).

OffMenu( )

Disables a particular menu or menu item.

OffMenu(Window, MenuNumber);

Registers a0—Window; d0—MenuNumber

Window Pointer to the Window structure of the window to which the menu belongs.

MenuNumber A menu number generated with the macros SHIFTMENU( ), SHIFTITEM( ), and SHIFTSUB( ) or returned by a MENUPICK IntuiMessage.

Note: OffMenu( ) will disable an entire menu, a single item, or a single subitem, depending on what MenuNumber indicates. When a menu, an item, or a subitem is disabled, it will appear ghosted and cannot be selected by the user.

See also OnMenu( ).

OnGadget( )

Activates a gadget.

OnGadget(Gadget, Pointer, Request);

Registers a0—Gadget; a1—Pointer; a2—Requester

Gadget Points to the Gadget structure which represents the gadget we are enabling.

Pointer A pointer to the graphics element the gadget appears in (a pointer to a Window structure if this gadget appears in a window).

Requester Points to a Requester structure if this is a requester gadget. Otherwise, set this to NULL.

Note: When a gadget is enabled, it will be displayed normally (that is, not ghosted), the GADGDISABLED flag will be cleared, and the user will once again be able to select the gadget.

See also OffGadget( ).

OnMenu( )

Activates a menu or menu item.

OnMenu(Window, MenuNumber);

Registers a0—Window; d0—MenuNumber
Window Pointer to the Window structure of the window to which the menu belongs.
MenuNumber A menu number generated with the macros SHIFTMENU( ), SHIFTITEM( ), and SHIFTSUB( ) or returned by a MENUPICK IntuiMessage.

Note: OnMenu( ) will enable an entire menu, a single item, or a single subitem, depending on what MenuNumber indicates. When a menu, an item, or a subitem is enabled, it will appear ghosted and cannot be selected by the user.

See also OffMenu( ).

OpenScreen( )

Opens a new screen.

struct Screen *OpenScreen(NewScreen);

Registers a0—NewScreen
NewScreen A pointer to a NewScreen structure.

Note: Returns NULL if the screen could not be opened. Otherwise, it returns a pointer to the Screen structure associated with the new screen.

See also CloseScreen( ).

OpenWindow( )

Opens a new window.

struct Window *OpenWindow(NewWindow);

Registers a0—NewWindow
NewWindow A pointer to a NewWindow structure.

Note: Returns NULL if the window could not be opened. Otherwise, it returns a pointer to the Window structure associated with the newly opened window.

See also CloseWindow( ).

RemoveGadget( )

Takes a gadget out of a window or screen.

USHORT RemoveGadget(Pointer, Gadget);

Registers a0—Pointer; a1—Gadget

pointer Pointer to the Window or Screen structure from which the gadget is to be removed.

gadget Pointer to the Gadget structure which is being removed from the gadget list.

Note: Will return the ordinal position of the gadget it removed. If the gadget wasn't found or if it was searching in the wrong list (that is, pointer was bad), then it will return −1.

See also AddGadget( ).

SetMenuStrip( )

Installs a menu strip to a window.

SetMenuStrip(Window, Menu);

Registers a0—Window; a1—Menu

Window A pointer to the Window structure of the window you want to attach the menu to.

Menu A pointer to the first menu in the menu system you've built.

See also ClearMenuStrip( ).

ViewAddress( )

Returns the address of the Intuition View structure.

struct View *ViewAddress( ); /* takes no parameters */

Registers None

Note: The view address is useful when using Kernel graphics primitives.

See also ViewPortAddress( ).

ViewPortAddress( )

Returns the ViewPortAddress of a particular window.

struct ViewPort *ViewPortAddress(Window);

Registers a0—Window

Window Pointer to the Window structure of the window whose ViewPort we want to find.

Note: The ViewPort address is an important variable when using Kernel graphics primitives.

See also ViewAddress( ).

Appendix F

Selected Kernel EXEC Routines

AllocMem( )

Acquires a block of free RAM.

APTR AllocMem(Size, Type);

Registers	d0—Size; d1—Type
Size	The number of bytes needed in a block. The size of the allocated block will always be a multiple of eight bytes. This parameter will be rounded upward if necessary.
Type	Desired characteristics of RAM block: MEMB_PUBLIC: Accessible to all executing tasks. MEMB_CHIP: Accessible to the Amiga's I/O hardware (video, sound, disk). MEMB_FAST: Nonchip memory. MEMB_CLEAR: Memory should be initialized to zero bytes.

Note: Returns pointer to the block if successful; zero, otherwise. On Amiga systems with 512K or less memory, the distinctions concerning public, chip, and fast RAM do not appear to apply.

See also FreeMem( ).

CloseLibrary( )

Informs library manager that we no longer require access to a particular library.

CloseLibrary(Library);

Registers	a1—Library
Library	Pointer to a valid Library structure.

See also OpenLibrary( ).

FreeMem( )

Returns a block to the pool of free RAM.

FreeMem(MemPointer, Size);

Registers	a1—MemPointer, d0—Size
MemPointer	Address of block being freed.

Size — Number of bytes in block being freed. If Size is not a multiple of eight bytes, it will be rounded upward, giving the actual size of a block allocated with the same Size parameter.

See also AllocMem( ).

GetMsg( )

Gets the next available message from a message port.

struct Message *GetMsg(Port);

Registers a0—Port

Port — A pointer to a MsgPort structure.

Note: Returns NULL if there was no message to receive. Otherwise, it returns a pointer to the Message structure which was sent.

See also PutMsg( ); ReplyMsg( ); WaitPort( ).

OpenLibrary( )

Informs library manager that we require access to a particular library.

struct Library *OpenLibrary(Name, Version);

Registers a1—Name; d0—Version

Name — A string holding the name of the library we want to open.

Version — The minimum version number of the library we can accept. This feature doesn't seem to be implemented, but should be set to zero.

Note: Returns a pointer to a Library structure if access to the library is permitted. Otherwise, it returns NULL.

See also CloseLibrary( ).

PutMsg( )

Sends a message to a particular port.

PutMsg(Port, Message);

Registers a0—Port, a1—Message

Port — A pointer to the MsgPort structure where the message should be sent.

Message — A pointer to the Message structure which is being sent.

See also GetMsg( ); ReplyMsg( ); WaitPort( ).

ReplyMsg( )

Sends a message back to the sender.

ReplyMsg(Message)
Registers a1—Message
Message A pointer to the Message structure which is to be returned to the sender.

Note: This routine assumes that the reply port field of the Message structure points to the proper message port.

See also GetMsg( ); PutMsg( ); WaitPort( ).

Wait( )

Puts the process to sleep and waits for one or more signal bits from other processes.

Wait(Signal);
Registers d0—Signal
Signal The signal bits that we are supposed to wait for.

WaitPort( )

Waits for a message to appear at a given port.

struct message *WaitPort(Port);
Registers a0—Port
Port A pointer to MsgPort structure.

Note: Waits for a message to appear at port. When one does, it returns with a pointer to the message that was sent. It will not remove the message from the message port. You must do this yourself using GetMsg( );

See also GetMsg( ); PutMsg( ); ReplyMsg( ).

Appendix G

Selected Kernel Graphics Routines

Move( )

Changes the location of the graphics pen.

Move(RastPort, x, y);

Registers	a1—RastPort; d0—x; d1—y
RastPort	A pointer to a raster structure. This is pointed to by the RPort field of the Window structure.
x,y	A point in the raster; (x,y) is relative to the upper left corner of the window.

ReadPixel( )

Reads the color at a particular point in a raster.

int ReadPixel(RastPort, x, y);

Registers	a1—RastPort; d0—x; d1—y
RastPort	A pointer to a Raster structure. This is pointed to by the RPort field of the Window structure.
x,y	A point in the raster; (x,y) is relative to the upper left corner of the window.

Note: Returns the pen used to draw that particular pixel. If the pixel could not be read (if it is off the edge of the screen, for example), this routine returns −1.

RectFill( )

Draws a rectangular solid in the raster.

RectFill(RastPort, xmin, ymin, xmax, ymax);

Registers	a1—RastPort; d0—xmin; d1—ymin; d2—xmax; d3—ymax
RastPort	A pointer to a Raster structure. This is pointed to by the RPort field of the Window structure.
xmin, ymin, xmax, ymax	The coordinates of the upper left and lower right corner of the rectangle to draw.

SetAPen( )

Sets the color register to use for APen drawing.

SetAPen(RastPort, pen);

Registers	a1—RastPort; d0—pen
RastPort	A pointer to a Raster structure. This is pointed to by the RPort field of the Window structure.
pen	The color register to use. This must be a value from 0 through 255.

Text( )

Draws characters.

int Text(RastPort, string, stringlength);

Registers	a1—RastPort; a0—string; d0—stringlength
RastPort	A pointer to a Raster structure. This is pointed to by the RPort field of the Window structure.
string	The string we are trying to write out.
stringlength	A count of the number of characters in the string.

Note: Drawing is started at the current position.

See also Move( ).

WritePixel( )

Draws a pixel of color APen at a particular point on a particular raster.

WritePixel(RastPort, x, y);

Registers	a1—RastPort; d0—x; d1—y
RastPort	A pointer to a Raster structure. This is pointed to by the RPort field of the Window structure.
x,y	A point in the raster; (x,y) is relative to the upper left corner of the window.

Appendix H

Selected DOS Library Routines

Close( )

Ends access to contents of a file.

Close(FileHandle);

Registers d1—FileHandle

FileHandle Handle to a file obtained calling Open( ). A program should close all files that it opens.

See also Open( ).

CurrentDir( )

Changes the current DOS directory and returns old directory.

APTR CurrentDir(NewDir);

Registers d1—NewDir

NewDir A pointer to a Lock structure describing the directory which will become the default directory for disk operation.

Note: Returns a lock to the previous current directory.

See also Lock( ).

DupLock( )

Makes a copy of a Lock structure.

APTR DupLock(Lock);

Registers d1—Lock

Lock Pointer to lock to be duplicated. Only nonexclusive (ACCESS_READ) locks can be duplicated.

Note: Returned value is pointer to copy.

See also Lock( ).

Examine( )

Gets information about a file or directory.

int Examine(Lock, FIB);

Registers d1—Lock; d2—FIB

Lock Pointer to a Lock structure indicating the file to be examined.

FIB Pointer to a FileInfoBlock structure. Examine will initialize it and store the information here.

Note: The information returned includes name, file length, and whether it is a file or a subdirectory. If a value of zero is returned, no information could be found.

See also Lock( ), ExNext( ).

ExNext( )

Gets information on the contents of a directory.

int ExNext(Lock, FIB);

Registers d1—Lock; d2—FIB

Lock Pointer to a Lock structure indicating the directory to be examined.

FIB Pointer to a FileInfoBlock structure, initialized by a previous call to Examine.

Note: Successive calls to ExNext will return information on each member of a directory. When a value of zero is returned, the end of the directory has been reached.

See also Lock( ), Examine( ).

Input( )

Returns file handle for the default input device.

ULONG Input( );

Registers None

Note: The file handle returned by Input will usually be associated with the CLI window, unless input has been redirected from the command line. Input will return zero if the calling program was not run from the CLI.

See also Output( ).

Lock( )

Requests access to a file or directory.

APTR Lock(Name, Access);

Registers d1—Name; d2—Access

Name Pointer to a null-terminated string containing the name of the file or directory to be accessed.

Access Lock can either be exclusive (ACCESS_WRITE) or non-exclusive (ACCESS_READ).

Note: Returns a pointer to a Lock structure which can be used to set the default directory or to catalog a directory.

See also DupLock( ), Examine( ), UnLock( ).

Open( )

Begins accessing the contents of a file.

ULONG Open(FileName, OpenMode)

Registers	d1—FileName; d2—OpenMode
FileName	Pointer to a null-terminated string containing the name of the file to be accessed.
OpenMode	Indicates whether a new file should be created if FileName cannot be found (MODE_NEWFILE) or not (MODE_OLDFILE).

Note: Returns a file handle which can be used for Read( ) and Write( ) operations.

See also Close( ), Read( ), Write( ).

Output( )

Returns file handle for default output device.

ULONG Output( );

Registers None

Note: The file handle returned by Output will usually be associated with the CLI window, unless output has been redirected from the command line. Output will return zero if the calling program was not run from the CLI.

See also Input( ).

Read( )

Gets data from a file or device.

LONG Read(FileHandle, Buffer, Count);

Registers	d1—FileHandle; d2—Buffer; d3—Count
FileHandle	Handle of the file to be read.
Buffer	Pointer to buffer to receive data.
Count	Number of bytes desired.

Note: Read returns the actual number of bytes that were read. If an error has occurred, it will return −1.

See also Open( ), Write( ).

UnLock( )

Releases access to a file or directory.

UnLock(Lock);

Registers	d1—Lock
Lock	Pointer to a Lock structure created by Lock or DupLock.

Note: If locks are not freed by UnLock, the RAM which they occupy cannot be reused.

See also Lock( ), DupLock( ).

Write( )

Sends data to a file or device.

LONG Write(FileHandle, Buffer, Count);

Registers	d1—FileHandle; d2—Buffer; d3—Count
FileHandle	Handle of the file to be written to.
Buffer	Pointer to buffer containing data.
Count	Number of bytes in buffer.

Note: Write returns the actual number of bytes that were written. If an error has occurred, it will return −1.

See also Open( ), Read( ).

Appendix I

Fast Floating Point Functions

Functions in "mathffp.library"

SPFix(a)	convert FFP to integer
SPFlt(a)	convert integer to FFP
SPCmp(a,b)	compare (equivalent to SPTst(a−b))
SPTst(a)	test (signum)
SPAbs(a)	absolute value
SPNeg(a)	negate
SPAdd(a,b)	addition
SPSub(a,b)	subtraction (b−a)
SPMul(a,b)	multiply
SPDiv(a,b)	divide (b/a)

Functions in "mathtrans.library"

SPFieee(a)	convert from IEEE format
SPTieee(a)	convert to IEEE format
SPAtan(a)	arctangent
SPSin(a)	sine
SPCosine(a)	cosine
SPTangent(a)	tangent
SPSincos(&b, a)	sine of *a* (cosine returned in *b*)
SPSinh(a)	hyperbolic sine
SPCosh(a)	hyperbolic cosine
SPTanh(a)	hyperbolic tangent
SPExp(a)	exponential
SPLog(a)	logarithm
SPPow(a,b)	raise *b* to the *a*th power
SPSqrt(a)	square root

Index

Programming the Amiga

Charles Brannon
Sheldon Leemon
Arlan Levitan
Dan McNeill
Chris Metcalf
Philip Nelson
C. Regena
Marc Sugiyama
Tim Victor

COMPUTE!'s Amiga Programmer's Guide is a comprehensive, detailed guide to programming the Amiga from Commodore. It is the reference you'll want next to you as you program. Whether you write in BASIC, C, or machine language, you'll find lots of useful information to help you tap the powerful features of the Amiga. Here's just a sample of what you'll find inside:

- A complete BASIC reference section with dozens of program examples.
- How to write C and machine language programs using Intuition, Kernal, and DOS routines.
- Comprehensive explanations of how to program sound and graphics from BASIC.
- Sample programs that illustrate how to create windows in C and machine language.
- A quick disk copy utility written in 68000 machine language.
- Mandelbrot, a C program that creates beautiful designs on your Amiga screen, including all the C source code to study and learn from.
- An easy-to-use AmigaDOS command summary.
- Numerous reference charts and tables.

Written clearly and concisely, *COMPUTE!'s Amiga Programmer's Guide* is packed full of useful information for both beginning and experienced programmers. It's the guide you'll find yourself referring to over and over again.

ISBN 0-87455-028-9

COMPUTE!'s AMIGA Programmer's Guide

COMPUTE! Books