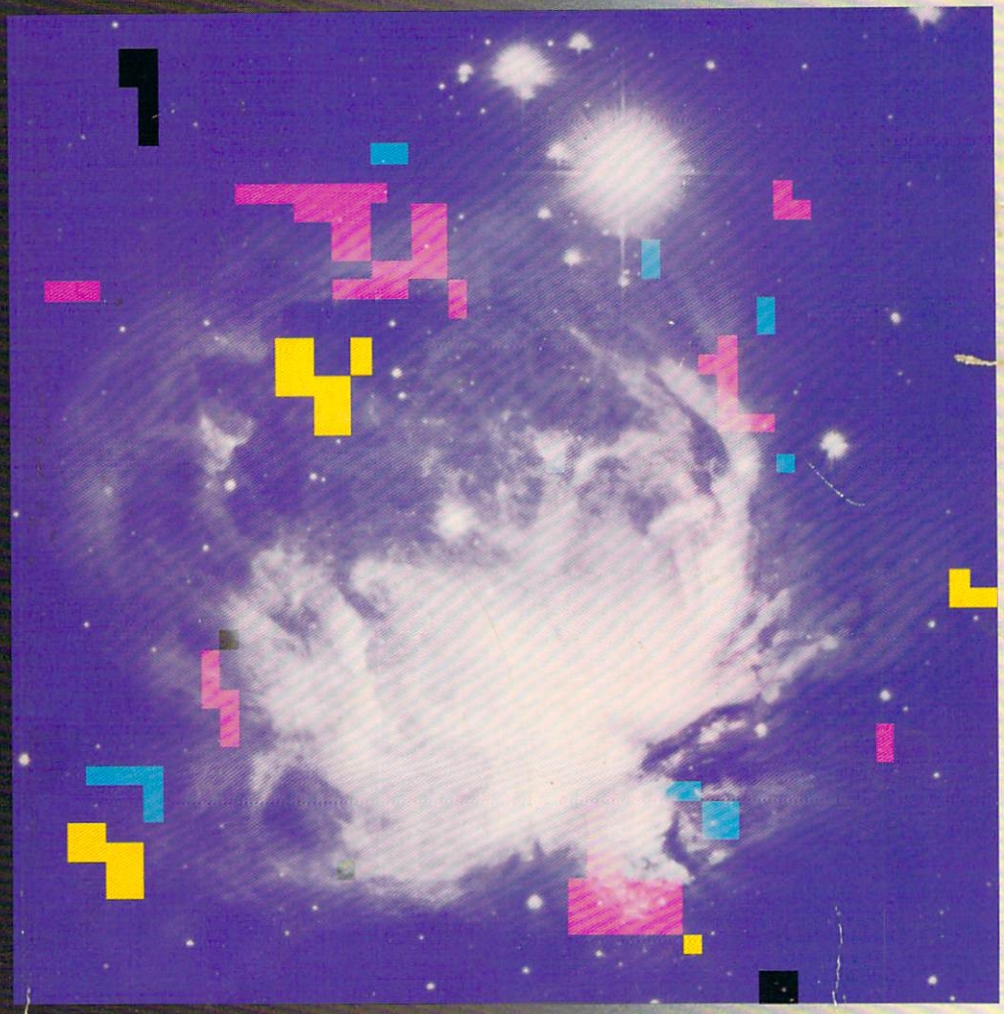
SYBEX

PROGRAMMER'S GUIDE TO THE AMIGA™

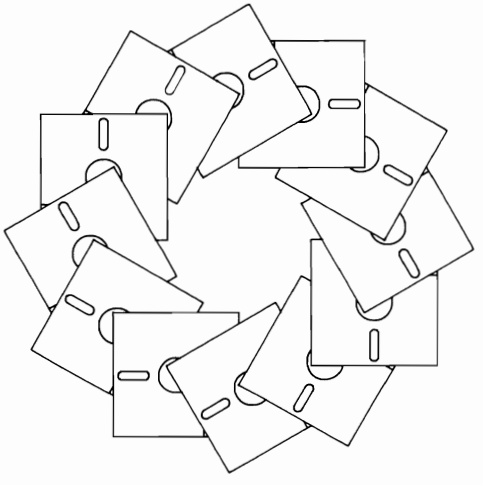
Robert A. Peck



SYBEX

Programmer's Guide to the Amiga

Programmer's Guide to the Amiga™



Robert A. Peck

SYBEX®

San Francisco • Paris • Düsseldorf • London

Book design by Jeff Giese
Cover art by Thomas Ingalls + Associates

Library of Congress Card Number: 86-63749
ISBN 0-89588-310-4
Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

To my wonderful wife, Andrea. I couldn't have done any of this without you.

ACKNOWLEDGMENTS

I wish to thank the management and staff at SYBEX for the effort that they put into producing this book. I particularly want to acknowledge Dr. Rudolph Langer and Karl Ray, who gave me the chance to tell programmers more about the Amiga. My editor, Valerie Robbins, provided help and guidance. It is through her efforts that this project finally came together.

The following people put a lot of effort into this book and their work is greatly appreciated: Olivia Shinomoto, word processing; Cheryl Vega, typesetting; Suzy Anger, proofreading; Julie Bilski, paste-up; and Paula Alston, indexing.

Thanks, too, to the crew at Hewlett-Packard for their support and encouragement during the past few months, and for the chance they gave me to work on an interesting and stimulating project.

And finally, hello to all of the Busy Guys who created the Amiga and to all of the software developers who worked on the early systems. Your efforts have provided a truly fine machine for all of us to enjoy.

Table of Contents

PREFACE

When writing the *Amiga ROM Kernel Manual*, we split the system into various functional areas. One section of the manual covered the system executive, another covered the graphics, others the sound, animation, math, devices, and so on. This split along functional lines provided one way of looking at the system. In the tutorial section of the manual, we tried to provide a more detailed view of how the various pieces fit together. This tutorial approach usually began by describing the data structures that the routines used, and ended with a functioning example that a reader could type in and try.

When I began this book, the *Programmers' Guide to the Amiga*, I felt that it should provide a different way of looking at the Amiga. My experience with user networks showed that examples were often much more effective than text at getting the point across. If a fully commented example shows exactly how a particular software feature can be used, it provides the greatest return. Thus, this book came about with the intention to provide as many short illustrative programming examples as possible. If a feature is described in this book, "there had better be an example to back it up."

Wherever possible, the examples are complete in themselves. However, sometimes, to simplify things a little, parts of a prior example or some form of setup routine is specified. You may have to compile a program segment, then link with the setup routine to complete the example.

The book begins by showing you how to make the Amiga do the things that other computers can do. Along the way, while doing those things, I've included some of the Amiga-specific functions—taking advantage of multitasking and so on. This book, as it goes along, provides program examples that supplement the *Amiga ROM Kernel Manual*, the *AmigaDOS Developers Manual*, and the *Intuition Manual*. Together, all of these pieces comprise the Amiga system software and that's what you're trying to understand. Along the way, since you'll most likely be using Amiga C to compile the programs, you'll encounter a few of the Amiga C functions.

C language compilers often come with some form of an interface library that provides, for example, character and file input and output routines. This book uses these language- specific interface routines as sparingly as possible. A reader using a C compiler other than Amiga C should therefore have little difficulty adapting the examples to another compiler. In the back of the book, to make compiler adaptation even easier, Appendix B includes a description of the ways in which Amiga C interfaces with the system software.

The designers and developers of the Amiga system software provided many routines to make the programming job easier. For example, most of the Amiga hardware is managed in a very efficient manner by system routines. In most cases, it isn't necessary to store data directly into system registers to cause a certain effect to occur. Nor is it necessary to plumb the depths of the hardware manual to discover how to perform a desired function. Using system software, you'll find that you can tell the Amiga what effect you want to accomplish and how it is to be done, and the system software will manipulate the hardware and memory bits to accomplish the task.

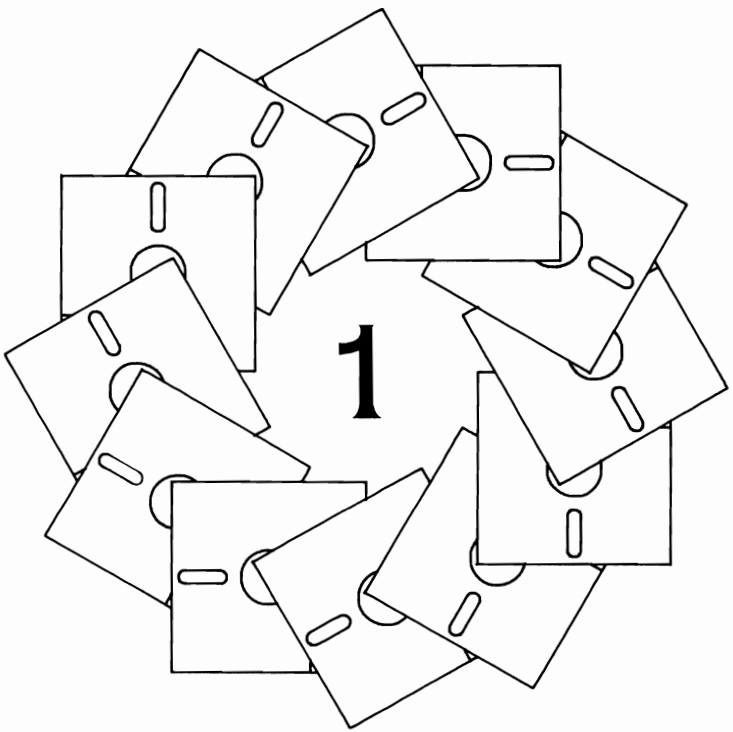
To keep things simple, wherever possible, I concentrate primarily on the end effect rather than on the underlying control registers or the contents of the data structures, unless they have an immediate bearing on what is being done. I hope that you'll find this method useful.

To keep things from getting too bogged down with details, this book concentrates on getting you up and running on the Amiga. Instead of trying to describe every option of every routine and every data structure, I've kept things as simple as possible and tried to show what programmers will likely encounter when working on the Amiga.

All of the examples in this book have been compiled using Amiga C version 1.1 and run under system software version 1.1 or 1.2. Please note that any references in the listings to include "myintuition.h" can be replaced by two include statements for the files "exec/types.h" and "intuition/intuition.h." If you don't want to type the listings, a complete disk is available. Please see the coupon in the back of the book.

Good luck.

Overview



The Amiga contains specialized hardware to help the main processor perform many functions. Some of this hardware handles the transfer of data to and from the disk. Some of it handles the joystick and mouse ports. Another piece of hardware handles the keyboard, and yet another handles the screen.

Although you can go directly to the hardware to control the Amiga, for any application, you will find it easier to utilize the system software routines (also called functions). These routines—of which there are over 300—provide easy access to the hardware and a consistent method of controlling various special features of the system.

AMIGA SOFTWARE AND HARDWARE HIERARCHY

Figure 1.1 shows a block diagram of the Amiga system software, known as Kickstart. As you can see, various levels of the system software are built upon each other. At the top of the figure is a type of applications program, that is, something that interacts with the user. At the bottom of the figure is the hardware itself. A programmer uses entry points throughout the system software to get a job done. Which of these entry points are used depends on the complexity of the application.

The Bottom Level

The direct interface to the system hardware is performed by several sets of system interface routines.

The set of routines that controls how the 68000 is utilized is called Exec. Exec shares the 68000 among many different programs (tasks) that might be loaded into memory at the same time; thus, the Amiga is capable of multitasking. Exec also handles allocating memory for these programs to use and handles interrupts that are generated by the special-purpose hardware or by system or applications software. Exec also maintains lists of tasks that are running or waiting for something to happen; lists of free areas of memory; lists of messages to which an application should respond; and lists of input events, such as mouse moves, timer ticks, keystrokes, and so on.

Software entities called devices control access to the disk, the keyboard, the gameport (the gameport handles mouse input), the audio system, and the serial and parallel ports. The graphics system software directly controls the graphics hardware and provides routines for

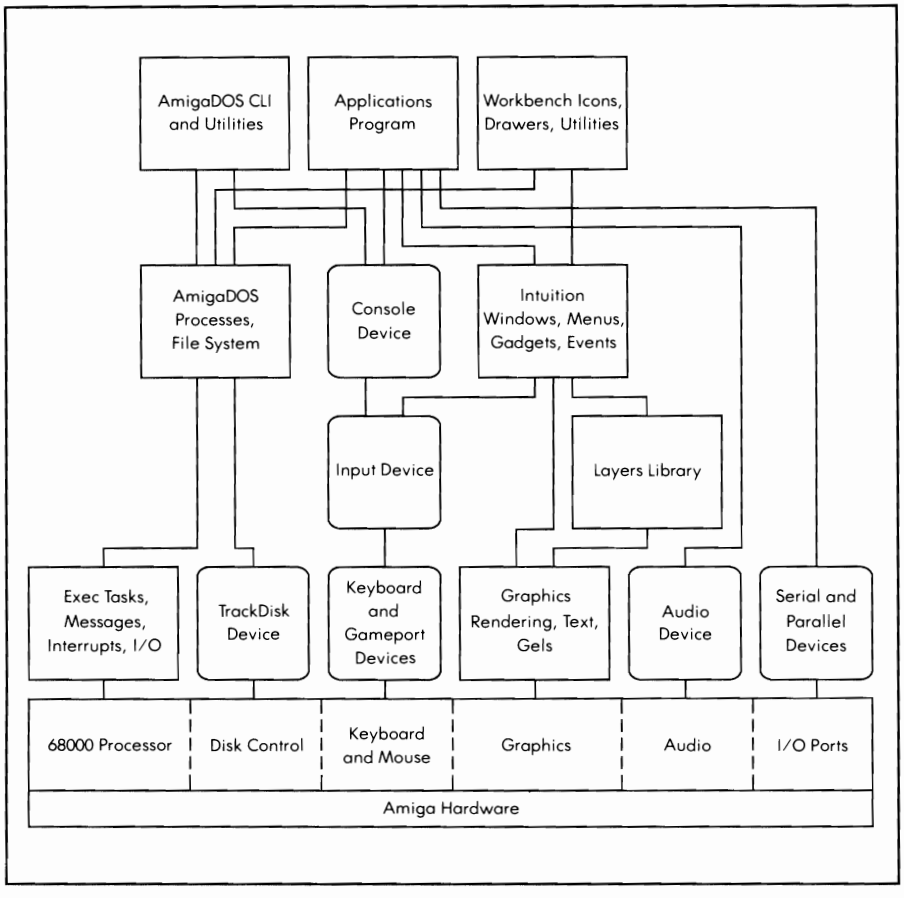


Figure 1.1: Amiga system software hierarchy

creating and controlling drawing and display areas, selecting colors and patterns, controlling screen resolution, manipulating graphics objects, and many more graphics functions.

The Second Level

On the next level above the hardware interface, you find the Input device and the Layers library. The Input device runs as an independent task in the system and merges together the input information (called input events) from the Keyboard device and the Gameport device (normally connected to a mouse) into a single input stream. The input stream, as shown in the figure, either feeds its events to Intuition or to a Console device.

The Layers library is built on top of the graphics system and provides routines for splitting a common drawing area into multiple, possibly overlapping layers that are the basis for building a windowing system. The Layers library includes routines for creating, deleting, and moving layers as well as for changing the front-to-back priority of layers and for manipulating the contents of these layers.

The Third Level

At the third level you find AmigaDOS, the Console device, and Intuition. AmigaDOS is a disk operating system with multiprocessing built in; it works with Exec to provide a means of sharing system resources, including sharing the central processor among cooperating tasks. AmigaDOS also provides a filing system (a way to access your data) and various utilities for manipulating that filing system and for initiating new processes.

Intuition is the multiscreen, multiwindow interface of the Amiga. Using Intuition routines, you can create a display that shows one or more screens of data. Each screen can have either 320 or 640 pixels of horizontal resolution, with either 200 or 400 lines of vertical resolution. Each 640-pixel-wide screen can have 16 different colors (out of a palette of 4,096 possible choices) on the display. Each 320-pixel-wide screen can have 32 different colors on the display.

Intuition is actually a library of functions. (See the heading Libraries of Functions later in this chapter.) You call these functions to create a visual interface for the user, splitting the screens into multiple subsections called windows, providing menus from which a user can choose options, and providing requesters (a special sort of a window that requests user input).

The Input device feeds input events into Intuition. Intuition, in turn, translates these events (keyboard key presses and releases, and mouse movements and button press/release combinations) into Intuition events to which you can respond as you choose. Intuition also filters these events for you so your program will see only those in which you are interested.

The Console device is a special device that you can attach to an Intuition window to make that window appear to be a computer terminal, similar to that used in the early days of personal computers. The Console device responds exactly like a piece of terminal hardware, capturing keystrokes that are directed to that window and typing responses in return.

Intuition acts as a traffic cop, making only one window active at one time. Thus, your keystrokes are sent to the window (and its console if it has one)

only when that window is activated by a mouse selection button press within the boundaries of that window.

The Top Level

At the top of Figure 1.1, you see three items: an applications program, the Workbench, and the CLI (Command Line Interface). Each of these presents its own form of interface to the user.

The CLI presents what might be called the traditional computer interface, i.e., something that appears to be a computer terminal. The CLI is actually an applications program that knows how to translate a typed line into a command that the computer is supposed to perform.

The Workbench substitutes a selection of icons and menu items wherever possible as a replacement for typing commands on the keyboard. It performs many of the same commands that could have been performed by the CLI. What it lacks in versatility, it makes up for in ease of use.

An applications program presents whatever form of interface the programmer desires, probably based on various underlying system software elements.

WHAT YOU NEED TO KNOW TO PROGRAM THE AMIGA

The things you need to know about to program the Amiga largely depend on the kind of applications program you are trying to develop. For example, you may want to create an applications program that takes advantage of the Amiga's fabulous graphics or sound capability. In this case, you'll need to know how to set up the data structure for graphics or sound, how to open their function libraries, and which functions to use for which purpose.

Or, you may want to use the multitasking to greatest advantage, perhaps by printing a document while you are editing another document, with yet another task retrieving electronic mail from a remote source for you. In this case you'll need to know about multitasking or at least how to ask AmigaDOS to start a separate process for you.

In general, you should understand the operation of the CLI and the Exec system in addition to the specialized system software you'll need for your applications programs, because each application needs a small amount of bookkeeping, so to speak, before you can use graphics or any other specialized Amiga system functions.

Libraries of Functions

In the description of the various levels of the Amiga system, the term *library* was used. A library is a collection of functions that are related to one another in some way. The Amiga designers gathered together related functions into libraries to ensure that programs written for one version of the Kickstart disk would be as compatible as possible with newer versions of Kickstart. To allow this compatibility, they had to make sure that programs could always find the system functions, no matter how the system had changed from version to version.

To accomplish this, the designers defined a Library data structure that contains, among other things, a table of jump instructions and function addresses, such as

```
            JUMP FunctionN
            ... ...
            JUMP Function3
            JUMP Function2
            JUMP Function1
LibBase     <start of Library data structure in memory>
            <rest of Library data structure>
```

When you cold start the Amiga, the various system library groups are sought in the Kickstart memory and are copied into RAM where they can be modified later if necessary. Then each of the libraries that is found is added to the system library list.

When the system powers up, a library can be positioned anywhere in the system memory. So, for your program to access routines in a library, it must declare a specific variable name—the library base address. For example, here is the code that is required to access an Intuition library function, where IntuitionBase is the base address of the Intuition library:

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"

struct IntuitionBase *IntuitionBase;                    /* must be global */
extern struct Library *OpenLibrary();

main()
{
IntuitionBase = (struct IntuitionBase *)
                OpenLibrary("intuition.library",0);
```

```
if(IntuitionBase == 0)
{
   printf("Intuition won't open!\n");
}
/* more program material... */
```

When you compile a program (see Appendix A), you link it with a file called amiga.lib, which adds special library function interface code to your program.

For a given library function, amiga.lib does the following:

- Saves the appropriate registers so that the system can continue running when the function returns
- Loads a register with the base address of the library in which it is located
- Sets up the registers with this function's parameters
- Jumps to a known offset address with respect to the base address of the library and enters the function
- Restores the registers to their state when the function was entered
- Returns a value where necessary

If you fail to declare the library name that relates to the function, the linker will tell you

```
_<someLibBase> undefined
```

If you declare the library base variable but fail to open the library before you try to use it, your program will crash.

If you both declare the library base variable and successfully open the library, you should be able to access the routines just fine. Appendix A of the *Amiga ROM Kernel Manual* lists the names of the base address variables as well as the libraries with which each is associated. Again, if you forget, the Amiga linker will tell you which libraries you must declare and open.

The dos.library and exec.library are opened automatically for you by the startup code that you link with your program (AStartup.obj or Lstartup.obj). Therefore, you can call AmigaDOS functions and Exec functions without explicitly opening a library to access them. There is more information on libraries in Chapter 3.

Programming in C

The examples in this book use the C language in part because there are at least two C compilers available for the Amiga, each with full support for the Amiga definition files (known as the Include files). In addition, all of the system data structures are relatively easily described in C, and a large percentage of the development of the Amiga operating system was done in C.

When you use C or Pascal or any other high level language, you will link the output of your higher level code with a library of routines that adapts the parameter passing conventions of that higher level language to the parameter passing conventions of the Amiga system code.

Programming in Assembly Language

If you use assembly code to call system routines, you should be aware of the Amiga's use of the registers of the 68000. First, registers D0, D1, A0, and A1 are always treated as scratch registers. They might be used to hold an input quantity. However, the contents of these registers are not guaranteed to be saved or restored by any of the system routines. The system saves and restores the contents of all of the other registers.

Functions that return a value return that value in register D0. If a function must return more that one value, you should plan to return the address of a data structure that contains an array or structure of the results.

Only one register gets special treatment in the Amiga system: A6. A6 is never used to pass parameters. The name of this register is SysBase. It contains the address of a function vector table that, in turn, contains the current addresses of various system functions. When a function is called, Exec jumps through the addresses contained in the table to get to the routine. Thus, a programmer can modify the table to make the system functions perform different operations or insert debugging or profiling code in line with the system functions to better analyze program performance.

You can find more details about programming in assembly code in the *Amiga Programmer's Handbook* by Eugene P. Mortimore (SYBEX, 1987), where all the registers used for each function call are provided.

The Include Files

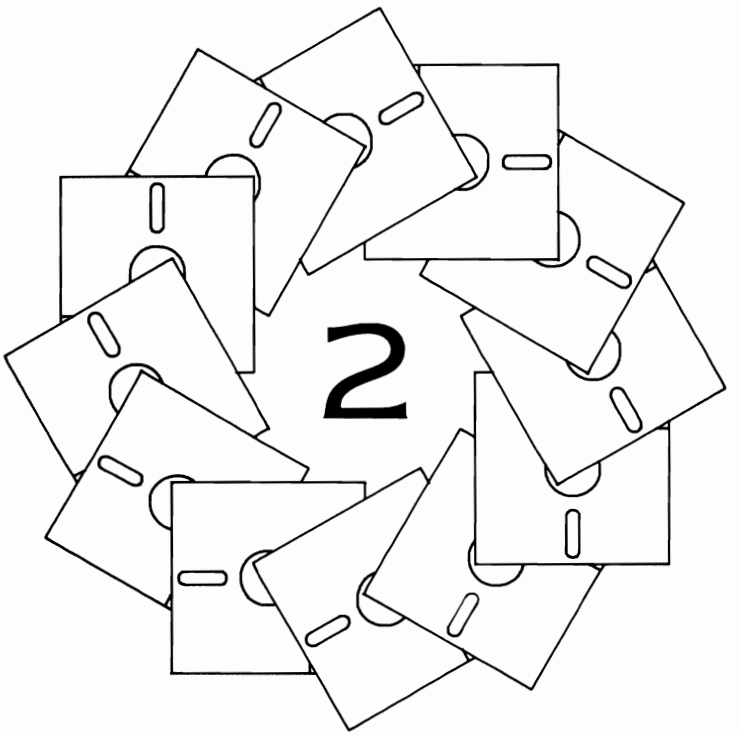
The Include files on your C compiler disk and Assembler disk provide definitions of the system constants and data structures. You will find references to these various Include files—having names ending in .h (for C)

or .i (for assembler)—throughout the book. These files are not listed in this book because the reader will also have obtained the *Amiga ROM Kernel Manual*, which provides listings of all the Include files. The purpose of this book is to supplement rather than to supplant that information.

You'll find many of the system routines described in the pages that follow in a guided tour of the system software. This book shows you what has to happen before a routine can actually run and suggests why you might want to use a particular routine. I hope that the tutorial approach is useful to you. Note that although many of the program examples in the book are complete, you will likely want to have your *Amiga ROM Kernel Manual* handy should you decide to modify them.

Well, on to programming.

AmigaDOS



Although the Amiga has an icon-based interface (the Workbench), it can also perform all of the functions that you would find on a terminal-based computer. This chapter describes the terminal-based features of AmigaDOS. You should find this chapter covers familiar ground. It demonstrates the following:

- Printing a line to the CLI
- Command-line argument passing under AmigaDOS
- Getting a line from the CLI
- Opening and closing files
- Reading and writing files
- Other file-related functions
- Opening a new custom CON window for I/O
- Opening a new custom RAW window for I/O
- Using AmigaDOS commands and utilities

All of the programs in this chapter must be started from the CLI because each outputs status information to the CLI window from which it is started. There are two different ways to start a program on the Amiga: by running a program from the CLI, or by opening the program's icon (if available) from the Workbench.

On the Amiga, to use any of the built-in system routines, it is necessary to open the library in which the routine resides. You will see the OpenLibrary calls used in other chapters for this purpose. This chapter uses no calls to OpenLibrary because the DOS library is opened automatically for you when the startup code (either AStartup.obj or LStartup.obj) is linked with your program. The startup programs open the DOS libarary for you. Thus, the library interface code need not be included in these examples. For other library routines described later in this book, you'll always find the library interface code (initially described in Chapter 3). Additionally, if you should choose later to use your own custom startup code, it will have to include a call to open the DOS library if you want to use any of the AmigaDOS functions.

PRINTING TO THE CLI

It is traditional that the very first program you find in a book on C or Pascal programming should be one that outputs the words "Hello

world." Rather than break with tradition, here is that same program for the Amiga. As you can see, it is neither long nor complicated to do this on the Amiga.

```
/* hello.c */

main( )
{
   printf("Hello world\n");
}
```

The directions for compiling this program and most of the other programs in this book are summarized in Appendix A. Use the standard version of the makesimple program (on your Amiga C disk, in the examples directory) to compile this program. When you compile and run this program, it prints the line "Hello world" to the CLI window, then it exits.

Assuming that you have compiled hello.c into an executable file named hello, you can start this program from the CLI by typing

```
hello
```

COMMAND-LINE ARGUMENT PASSING UNDER AMIGADOS

The startup files (Astartup.obj and Lstartup.obj) both provide command-line passing similar to that provided by Unix or MS-DOS in that a C language program can obtain the arguments (i.e., parameters) that appear on the command line along with the program name. For example, if you have designed a program called argecho, and you type the command line

```
argecho firstarg secondarg thirdarg
```

then on entry to your program, the arguments-count will be three, and the arguments array (argv[0],argv[1],argv[2]) will point to null-terminated strings each in turn containing one of your command-line arguments. Listing 2.1 is a program called argecho that types the command-line arguments it receives.

REDIRECTING STANDARD INPUT AND OUTPUT

On the command line, you can use Unix-like redirection symbols to ask AmigaDOS to use something other than the current CLI window for

```
/* argecho.c */

#include "exec/types.h"
#include "libraries/dosextens.h"

main(argc,argv)
int argc;
char **argv;
{
     int j;
     for(j=0; j<argc; j++)
     {
         printf("Argument number %ld is %ls\n",j,argv[j]);
     }
}
```

Listing 2.1: The argecho program

standard input or standard output. (Standard, in this sense, means the default, i.e., it is used if no other is supplied.) You do this by placing the redirection symbols < and > on the command line. Unlike Unix, however, they must come immediately after the command itself, preceding any of the arguments that you wish to pass to your program.

Here are a couple of examples of redirection, using the argecho program and a hypothetical program named mycopy. This line copies from standard input (stdin) to standard output (stdout), assuming the existence of a program called mycopy that simply reads characters one at a time from the standard input and prints them one at a time to the standard output:

```
mycopy < sourcefile > destfile
```

This means "mycopy (take stdin from) sourcefile (put stdout into) destfile."

The following line puts the result of argecho into a file named echofile:

```
argecho > echofile arg1 arg2 arg3 arg4
```

In this example, the redirection symbol and destination file name are stripped from the command line by AmigaDOS and your program receives only

```
argecho arg1 arg2 arg3 arg4
```

This means that there are four arguments that will be listed in a file named echofile.

When you use redirection in a command line, AmigaDOS actually opens a *file handle* that it uses internally for handling the data stream for input or output. When your command is carried out, this file handle is

used to close the file automatically. Your program need never become aware that this redirection happened at all; it simply took information from its standard input stream and sent information to its standard output stream and AmigaDOS handled the rest. The notations stdin and stdout are actually defined in the startup files as file handles. If you do not redefine the pathway to be used for stdin or stdout, they are equated to file handles for the current console window.

The next section goes into direct use of these file handles, pointing out where you might want to deliberately create and use file handles to accomplish a particular task.

FILE HANDLES

A file handle is a kind of pointer that you obtain from AmigaDOS when you open a file for reading or writing. It is subsequently used for things such as the Execute command redirection parameters, the Write function, the Read function, and finally the Close function that completes the access to a file.

The printf function, a standard function in Amiga C, is very commonly used to send output to the currently assigned standard output path. There is a second way that you can send output to the current CLI window. You do this by opening the current window for input and output using a file handle.

Note that file handles under Amiga C and file descriptors for AmigaDOS operations are different. You cannot, for example, pass a file handle obtained by an Amiga C fopen command to an AmigaDOS function, nor can you pass an AmigaDOS file descriptor obtained by the AmigaDOS Open function to an Amiga C function that is supposed to do file operations. You can choose to use either the Amiga C I/O functions or equivalent AmigaDOS I/O functions. Simply be sure to use the appropriate file handle when calling any function.

If you utilize the Amiga C functions for your I/O wherever possible, you will find it easier to translate your programs to run on other systems. However, by directly using the AmigaDOS I/O function, you might create a program that runs more efficiently on the Amiga.

The Amiga C manual describes the standard I/O functions such as putchar and getchar. This book shows how to use the AmigaDOS functions such as Read and Write.

Opening a File

To gain access to a file, you need a file handle. You obtain a file handle—a pointer to a data structure that contains information about that file—by using the AmigaDOS Open function.

A call to the Open function to open a file takes this form:

```
filehandle = Open(pathname,accessmode);

struct FileHandle *filehandle, *Open( );
char *pathname;
int accessmode;
```

The Open function takes two parameters. The first parameter is a pointer to a string that defines the file name. The file name specifier can include the complete directory path name as shown in this example. If you specify only a name, the current directory is used.

The second parameter is either MODE_OLDFILE or MODE_NEWFILE. (These terms are defined in the dos.h Include file.) MODE_OLDFILE opens a file for read/write with the file pointer positioned at the beginning of the file. MODE_NEWFILE opens a newly created file with this name, again for read/write, with the pointer positioned at the beginning of the file. If you use MODE_NEWFILE on an existing file, the existing version is deleted (if deletable) and an empty file by this name is created. If the current version of the file is not deletable (see Protecting a File later in this chapter), the Open command fails.

Closing a File

You terminate activity on a file by closing the file. A call to the Close function takes this form:

```
Close(filehandle);

struct FileHandle *filehandle;
```

The Close function takes one parameter—the file handle returned by the Open function. It terminates access to the file and writes out any output that might have been internally buffered by AmigaDOS.

Reading from a File

You obtain information from a file by using Read. A call to the Read function takes this form:

```
actual_count = Read(filehandle,buffer,count);
```

You tell AmigaDOS where to locate the opened file by passing a file handle obtained from an Open command. You also specify the count of characters you wish to read and the address of the buffer memory into

which the characters are to be placed:

```
struct FileHandle *filehandle;
char *buffer;
int count, actualcount;
```

AmigaDOS returns the actual count of characters read.

If your call to Read was able to fulfill your request, actual_count will match count. If actual_count is equal to 0, then no characters were read and you have reached the end of the file. If actual_count is equal to −1, then there has been an error. You can get more information about the error from IoErr. The Amiga library function named getchar uses the Read function.

Writing to a File

You add new information to a file by writing to it. A call to Write takes this form:

```
actual_count = Write(filehandle,buffer,count);
```

You tell AmigaDOS where to locate the opened file by passing a file handle obtained from an Open command. You also specify the count of characters you wish to write and the address of the memory buffer from which the characters are to be written:

```
struct FileHandle *filehandle;
char *buffer;
int count, actualcount;
```

AmigaDOS returns the actual count of characters written.

If the actual_count is equal to −1, then an error has occurred. You can get more information about the error from IoErr. The Amiga library function named putchar uses the Write function.

CONSOLE I/O USING AMIGADOS FILE HANDLES

Following is a program that opens the current CLI window for input and output. As with hello.c, it also prints "Hello world," but this program uses a different method. In this example, the "*" notation means "use the current window to locate the standard input and output path." The notation "%ls" is the format-string descriptor that tells the function fprintf how to print the string. "%" says this is the start of formatting data; "l" says there is a 32-bit (i.e., LONG) pointer to where the string data is located; "s" says print the data as ASCII characters.

```
#include "libraries/dosextens.h"          /* this defines FileHandle */
extern struct FileHandle *Open();         /* declare the function type */
main( )
{
   struct FileHandle *dos_fh;
   dos_fh = Open("*",MODE_OLDFILE);       /* open the console */

   fprintf(dos_fh,"%ls","Hello world\n");          /* amiga.lib func */
   Close(dos_fh);
}
```

Opening a New Window for Output

In place of the current window, you could open a new console window for input and output by changing the specification string for the Open command. Here is the modified version of the previous program; this version opens a new window:

```
#include "libraries/dosextens.h"
extern struct FileHandle *Open( )
main( )
{
   struct FileHandle *dos_fd;
   dos_fh = Open("CON:10/10/500/150/New Window",
   MODE_NEWFILE);

   Write(dos_fh,"Hello world\n",13);
   Delay(300);                                    /* delay 6 seconds */
   Close(dos_fh);
}
```

The window type is CON, meaning that it is to work just like a console. (You'll find more about consoles in the next section.) The on-screen position is to begin at X,Y coordinates of 10,10 and it is to be 500 pixels wide and 150 pixels high. New Window names the window. The AmigaDOS Delay function (where delay is specified in fiftieths of a second) is used so that you can see the output before the window closes and disappears. This time the mode for Open is specified as MODE_NEWFILE because this window is not currently open.

Getting Input from a Console Window

The same console window that you opened for output can also be used for input. The new window becomes active as soon as it is opened (unless you click the mouse selection button on another window, of course). So if

you type on the keyboard, the key inputs will be directed automatically to the new window. Listing 2.2 is a modification of the previous program; this program requests a line of input from you.

If you try this program, you will see that only printable characters are accepted by this console window, and nothing gets printed into the original CLI window until you press Return. Function keys and arrow keys have no effect. Essentially, AmigaDOS is receiving your input and filtering it.

Because AmigaDOS is filtering the input, you can use the AmigaDOS simple input editing commands such as pressing Backspace to backspace the cursor to erase the character most recently typed or using the key combination Ctrl and X to restart the input line completely. When you have finished formulating an input line, you press Return and AmigaDOS passes this finished line to the program.

Getting Input without Pressing Return

In the previous example, the CON window accepts filtered input and waits until you press Return before it returns anything from the Read function call. If you have to respond to every single keystroke as it actually occurs, you can use a RAW window in place of a CON window.

Listing 2.3 is a program that reports each keystroke to the CLI as it occurs. If you press something other than a keyboard key (a function key, arrow key, or help key) you will see that more than one value is generated for each of the key presses. When you press Q, the sample program ends.

```
#include "libraries/dosextens.h"
extern struct FileHandle *Open();

main()
{
        char userinput[256];
        int howmany;
        struct FileHandle *dos_fh;

        dos_fh = Open("CON:10/10/500/150/New Window",MODE_NEWFILE);
        Write(dos_fh,
                "Please type an input line, then press RETURN\n", 45);
        howmany = Read(dos_fh,userinput,255);
        userinput[howmany] = '\0';
        printf("You typed %ld characters:\n",howmany);
        printf("and here they are:\n");
        printf("%ls\n",userinput);
        Close(dos_fh);
}
```

Listing 2.2: The getaline program

```
#include "exec/types.h"
#include "libraries/dosextens.h"
#define QUIT 0x51 /* the uppercase (shifted) "Q" key */

extern struct FileHandle *Open();

main()
{
        char userinput[256];
        int howmany,j;
        struct FileHandle *dos_fh;

        dos_fh = Open("RAW:10/10/500/150/New Window",MODE_NEWFILE);

        for(;;)
        {
                howmany = Read(dos_fh,userinput,255);
                userinput[howmany] = '\0';

                printf("just got %ld values:\n",howmany);
                /* note: if a user types quickly, more than */
                         /* key downstroke report will be sent */
                         /* within a single timing interval. */

                printf("value stream was: ");
                for(j=0; j<howmany; j++)
                {
                        printf("%lx ",userinput[j]);
                }
                printf("\n");
                if(userinput[0]==QUIT) break;
        }
        Close(dos_fh);
}
```

Listing 2.3: The reportraw program

It is possible to sense when the user presses or releases any key. However, this facility is not provided through AmigaDOS. To obtain a unique event identifier for each key downstroke or upstroke, use Intuition's reporting mechanism called the IDCMP (Intuition Direct Communication Message Port). This facility also lets you listen in on mouse movements and mouse button clicks. The IDCMP is covered in Chapter 5.

SENDING OUTPUT TO A PRINTER

AmigaDOS lets you send information to a printer connected to either the serial or parallel port. There are three possible AmigaDOS paths that you can use to transmit output to a printer:

- SER:—the serial port that is accessed through the Serial device
- PAR:—the parallel port that is accessed through the Parallel device

- PRT:—the printer port that can be either serial or parallel depending on the setting of the Amiga Preferences; it is accessed through the Printer device.

These are named devices that can be opened just like any AmigaDOS file, and the Open command returns a file handle that can be used for redirection or for receiving output from your program. Listing 2.4 is a program that opens the Printer device for output, then copies a file to the printer.

The command that is implemented in this program is similar to the combination of the following direct CLI commands:

```
JOIN filename1 filename2 morefilenames AS JoinedName
TYPE > PRT: JoinedName
```

Notice that the Open function could just as easily have used the Serial or Parallel device. In this case, the output would be directed through that particular path, with characteristics you have preset in Preferences (serial baud rate, parity, number of bits, and so on).

AmigaDOS provides methods of defining how a printer is supposed to react to various standard control codes. When you go through the Printer device, the control codes you send to the printer are translated according to the type of printer you have declared in Preferences. When you send output to the printer directly through the Serial or Parallel device, you will not get this translation; instead, the printer will see exactly what you send. You may find this capability useful for controlling a printer that may not be available in the current Preferences selections.

Chapter 6 provides more information about how printers are supposed to react to the various control codes and also explains how to access the Printer device directly rather than through AmigaDOS.

MORE FUNCTIONS FOR MANIPULATING FILES

Here are a few more functions that are commonly required for file manipulation:

- Seek—moves to a different position in the file or simply inquires about the current value of the file position pointer
- IsInteractive—determines if the file handle is associated with a Console device

```
/* printem.c */

#include "libraries/dosextens.h"
extern struct FileHandle *Open();

main(argc,argv)
int argc;
char *argv[];
{
    struct FileHandle *fh, *fh2;
    int datasize;
    int n;
    char buffer [256];
    n = 1;

    if(argc < 2)
    {
        printf("Usage: printem <filename1> [ more file names ]\n");
        exit(0);
    }
    fh = Open("PRT:",MODE_OLDFILE);
    if(fh == 0) exit(20);        /* printer wouldn't open */

    while(argc > 1)
    {
        fh2 = Open(argv[n],MODE_OLDFILE);
        if(fh2 == 0)
        {
            printf("%ls: file not found\n"argv[n]);
        }
        else
        for(;;)                  /* forever (till error, probably EOF) */
            {
            datasize = Read(fh2,buffer,256); /* read the file */
            Write(fh,buffer,datasize);       /* write the output */
            if(datasize < 256) break;        /* read less than asked? */
                /* should check IOErr() here to see if EOF */
        }
        Close(fh2);
        n++; argc--;
    }
    Close(fh);
}
```

Listing 2.4: The printem program

- WaitForChar—waits for a character from a Console device but only for a limited time

Finding a Position in a File

When AmigaDOS opens a file and returns a file handle, the file is positioned at the beginning. You can use the Seek function to find any position within the file. You can specify that position relative to the current position, relative to the beginning of the file, or relative to the end of the file. Seek returns the value of the current position. The file size is specified in bytes, so each position is a byte-position in the file.

A call to Seek takes this form:

```
currentposition = Seek(filehandle,position,relative_to_what);

struct FileHandle *filehandle;
int position;
int relative_to_what;
int currentposition;
```

To move to the very end of the file to append something, type

```
currentposition = Seek(filehandle,0,OFFSET_END);
```

To report the current position without moving in the file, type

```
currentposition = Seek(filehandle,0,OFFSET_CURRENT);
```

To move ten bytes farther than the current position in the file, type

```
currentposition = Seek(filehandle,10,OFFSET_CURRENT);
```

To return to the very beginning of the file, type

```
currentposition = Seek(filehandle,0,OFFSET_BEGINNING);
```

Determining whether Your Program Is Connected to a Terminal

You can determine if the standard input of your program is connected to a CLI by using the IsInteractive function. This function returns a non-zero value if your program was started from the CLI and a zero value if you started the program from Workbench or by using the Execute function. When you start a program from the CLI, you provide a window into which the user can type a reply. This means the program is *interactive* (can exchange data with the user). If the program is started from some other function, there will be no input window and the standard input is not interactive. This allows you to decide not to output instructions to the user, waiting forever for a keystroke that may never come. The call takes this form:

```
status = IsInteractive();

int status;
```

Waiting for a Character for a Specified Time

If (and only if) a file handle is connected to an interactive terminal (CLI), the WaitForChar function can be used to wait a specified time until a character becomes available. Perhaps you might want to flash the

screen or beep and repeat your message if a user doesn't respond. Or perhaps you want to exit the program if there has been no keystroke by a certain time.

You can specify both the file handle for the input and the length of time that the program should wait before returning to allow you to execute the next instruction. The value returned by WaitForChar is a Boolean value (True = nonzero, False = 0) that tells you whether there is a character waiting to be read. You can decide what to do from there. The call takes this form:

```
status = WaitForChar(filehandle,timeout);

BOOL status;
struct FileHandle *filehandle;
int timeout;
```

Note that you specify the time-out value in fiftieths of a second.

This is a friendly use of the Amiga multitasking system in that your task actually goes to sleep, waiting for either a key-received interrupt or a time-out interrupt. Other tasks are free to run while your task remains asleep waiting for these things to happen.

AMIGADOS DIRECTORY STRUCTURE

AmigaDOS implements a hierarchical filing system. You can think of this filing system as though the disk (called a disk volume) is a filing cabinet with folders and papers. The file folders are subdirectories of your disk volume. The papers, individually, are your data or program files. The folders can contain papers or other folders.

When you request a directory listing, AmigaDOS tells you which of the names in the directory belong to files and which are directories by using the notation (dir) following the name. The root directory is the topmost entry in a disk volume. Every other file is either part of the root or is contained in a subdirectory that is part of the root directory.

In AmigaDOS, the root directory of any current disk is specified by using a colon. If you type

```
CD :
```

it means "make the current directory (CD) the root of the current disk." If your current directory path is df0:test/mystuff, the command "CD:" makes df0: the current directory. If your current directory is df1:a/b/c, then "CD:" makes df1: the current directory. This command moves to the topmost level, or *root* of the current directory path. From this root

directory, you can move to a lower level in the hierarchy by changing the current directory to be one of the directories found in the root.

For example, if a dir command executed at the root level of a disk lists

```
tests (dir)
```

then you can make this the current directory by specifying

```
CD tests
```

or go directly to it from some other directory by specifying the complete path name to make the current directory:

```
CD df0:tests
```

You can refer to an AmigaDOS disk by volume name (for example, mydisk:testfiles/firsttry) instead of simply by disk designation (for example, df0:xxx or df1:yyy). AmigaDOS is smart enough to ask for the correct disk to be inserted into any drive before the operation is continued. The volume name of a disk is applied to it at the time you format the disk or at the time you use DISKCOPY or RENAME from the Workbench. Remember this facility as you examine the sections that follow.

If you are programming the Amiga, the above information about path names and volume names should already be familiar to you. However, it was provided here simply to remind you that you may find it necessary during a program to move around in the hierarchy of the file system to access various files. The section that follows explains how you can have your program move around for you.

Dropping into AmigaDOS

It is comforting to be able to type commands directly into a CLI window and have AmigaDOS execute them for you. When you program for AmigaDOS, you'll find that you can call the functions for deleting files and renaming files directly. Also, you can programmatically perform any of the utilities such as copying a file or changing a directory.

In addition to accessing filing system functions, you can actually perform CLI-style commands directly from within the programming system by using the AmigaDOS Execute command. Execute takes a command string exactly as you would have typed it into the CLI and executes it as though there were a CLI present.

Execute has two restrictions:

- The RUN command must be present in the directory that you assign to drive C with an ASSIGN statement.
- The command that you Execute must be either in the current directory or in the C directory.

Following is a program using the Execute command. Notice that the command string includes a redirection command to place the output of the command into a file.

```
/* execute.demo.c */

#include ''libraries/dosextens.h''

main( )
{
   int success;
   success = Execute(''dir > df0:dir.file'',0,0);
   if(success = = 0) printf(''I/O error %ld'',IoErr( ));
}
```

The program places a listing of the current directory into a file named dir.file on the drive df0. To see the results of this program in the CLI, type this line:

```
TYPE df0:dir.file
```

The Execute function takes three parameters. The first parameter is a command string that may optionally include a redirection command; that is, an arrow (<) to show where the standard input is to come from, and an arrow (>) indicating where the standard output is to be directed.

The second and third parameters are redirection file handles; they specify how to redirect the standard input and standard output of the command if there is no standard input or output redirection specified in the command string that is passed as the first parameter. A value of zero for both parameters causes AmigaDOS to assume that the standard input and output for this Execute command is to be the same as for the process that calls the function. Thus, a simple command such as DIR prints its output directly to the CLI window if the execute.demo program was started from the CLI.

Calling a Function versus Using Execute

As with all of the above utilities, calling the function directly instead of using the Execute function has both advantages and disadvantages. If you use the function itself rather than Execute, your program does not incur the overhead of the Execute command (the RUN function and the command itself) and needs neither the RUN nor the command itself in the C directory of your disk. However, calling the function directly does prevent you from using the wild-card features that are available with Execute.

Using the File Handle Returned by the Open Function

As mentioned earlier, you can have AmigaDOS manipulate file handles for you by using redirection on the command line, or you can open a file yourself and use the file handle that the Open command provides to work with that file. Here is yet another example of using command-line redirection:

```
DIR > dir.list
```

This executes the DIR command and places its output into a file named dir.list. Listing 2.5 is a program that does the same thing but doesn't use the redirection information in the command string; instead, a file handle is obtained by opening a file named dir.list on the internal drive. The end effect is identical. It simply demonstrates the use of the file-handle parameters in the Execute command. You'll see more uses of file handles later in this chapter.

Moving across Branches in the Directory Tree

The Execute example shown earlier simply executed a command affecting the current directory. You may find it necessary to go to another directory, or, for example, ask AmigaDOS to request that the user insert a different disk.

AmigaDOS provides a mechanism for moving around in its file system. This mechanism is called the *lock*. Getting a lock on a directory is simply a

```
/* execute.demo2.c */

#include "libraries/dosextens.h"
extern struct FileHandle *Open();
main()
{
        int success;
        struct FileHandle *outhandle;

        outhandle = Open("df0:dir.list",MODE_NEWFILE);
        if(outhandle == 0)
        {
                printf("I/O error %ld\n",IoErr());
                exit(20);
        }
        success = Execute("dir",0,outhandle);
        if(success == 0)
                printf("I/O error %ld\n",IoErr());

        Close(outhandle);        /* close the file */
}
```

Listing 2.5: The execute.demo2 program

way of telling AmigaDOS to refer all of your requests for file access to a particular directory. This directory can be the root directory of a disk of a particular volume name or any subdirectory within any disk. Once you have a lock on a directory, you can get information about that directory or information about the files within that directory. It is not necessary to use a lock just to open, read, or write files. The lock mechanism provides the gateway for a program to access and possibly modify data that is being maintained by AmigaDOS.

This directory-locking mechanism is necessary in a multiprocessing system. For example, if your program obtains a lock on a particular directory path, then AmigaDOS will prevent another process from deleting this directory or any of its parent directories while the lock is in effect. Because the correct operation of AmigaDOS depends on the proper use of this locking mechanism, if you do obtain a valid lock, you must unlock it before your program exits.

Listing 2.6 is a program that uses a lock to set the current working directory. This program is equivalent to using the cd <*somedirectory*> command from the CLI but is effective for your program instead. The program does not change the directory in which the CLI is working but simply changes the directory in which your own program will work. Instead of listing the directory in which the CLI is currently sitting, the my.cdir program moves its current working directory to the directory named "C" on the internal disk. The listing is generated to the CLI window.

If you wish to try the program, compile it and store the executable file as my.cdir. Type the command DIR to obtain a listing of the directory in which you are located. Type my.cdir and you'll get a directory listing of the df0:c directory instead. Type CD and you'll notice that the CLI from which you ran the my.cdir program has remained within the same directory in which you began and that only your program has moved around within the filing system to perform its job.

Climbing Around in the Directory Tree

There are several functions that you need to use to move around in the AmigaDOS directories:

- IoErr—returns the error value from the most recent AmigaDOS operation
- Lock—gains control of a specific directory path
- UnLock—releases control of a locked directory path
- CurrentDir—moves into a specific directory

```
/* my.cdir.c */

#include "libraries/dosextens.h"
extern struct FileHandle *Open();
extern struct FileLock *Lock(),*CurrentDir();
main()
{
        int success;
        struct FileLock *lock,*oldlock;
        struct FileHandle *myfile;

        /* get a pointer to a specific directory */
        lock = Lock("df0:c",ACCESS_READ);
        if(lock == 0)
        {
                printf("\nCan't get a lock!");
                exit(20);
        }
        /* move into that directory if the pointer is valid */
        oldlock = CurrentDir(lock);

        /* open a file and thereby get a handle for accessing it */
        myfile = Open("df0:dir.file",MODE_NEWFILE);
        if(myfile == 0)
        {
            printf("open did not work: %ld\n",IoErr());
            /* Move to original directory */
            Oldlock = CurrentDir(oldlock);
            UnLock(lock);
            exit(30);
        }
        /* execute a command, redirect output into the */
        /* file via the handle */
        success = Execute("dir",0,0);
        if(success == 0)
        {
            printf("execute error:  %ld\n",IoErr());
            /* Move to original directory */
            Oldlock = CurrentDir(oldlock);
            UnLock(lock);
            exit(40);
        }
        /* Move to original directory */
        Oldlock = CurrentDir(oldlock);
        /* close the file */
        Close(myfile);

        /* and cleanup by unlocking anything we locked */
        UnLock(lock);
}
```

Listing 2.6: The my.cdir program

- Examine—fills a data structure with information about a specific directory
- ExNext—fills a data structure with information about files in a directory
- ParentDir—moves into the parent directory

Reading AmigaDOS Error Codes

When you are using AmigaDOS functions, their calling sequences often specify

```
success = Function(parameters);
```

or

```
pointer = Function(parameters);
```

where a value of zero returned means that the function failed for some reason.

You can find out why the function did not succeed by reading the error code that AmigaDOS provides. This error code is accessed by the IoErr function. A call to IoErr takes this form:

```
error = IoErr();
```

The possible errors that you can encounter are listed in the Amiga C Include file called libraries/dos.h.

Locking Files and Directories

A call to the Lock function takes this form:

```
mylock = Lock(pathname_string,access_mode);
```

The Lock function takes two parameters. The first parameter is a string that specifies the directory path you wish to lock. The string can be a complete path including a volume name, a name of another directory in the current directory, or the null string (""). You'll see several uses of the null string parameter later in this chapter.

The second parameter is the access mode, specified as ACCESS_READ, also called SHARED_LOCK. If you specify a shared lock, then other processes can also read and write into this directory or file. The other access mode is called ACCESS_WRITE or EXCLUSIVE_LOCK. No other process can access this directory or file if you use this mode.

The Lock function returns a pointer to a FileLock data structure, which contains information that AmigaDOS will use later to access a file (if you lock a file) or to access items in a directory (if you lock a directory).

The Lock function has a lower overhead than the Open function. Thus, as a faster way of determining if a file exists, you can try to lock it. If the function returns with a valid value (any nonzero value), the file exists and you can then decide to open it. If Lock returns a zero value, the file was not found.

Unlocking Files and Directories

A call to the UnLock function takes this form:

```
UnLock(mylock);
```

The parameter is a pointer to a FileLock data structure. You must unlock anything that you lock so that AmigaDOS can continue to function correctly.

Moving from One Directory to Another

You use CurrentDir to move from one directory to another. The value of oldlock is provided to let you come back to wherever you were in the first place. A call to CurrentDir takes the form

```
oldlock = CurrentDir(mylock);
```

where mylock is a pointer to a lock that you obtained from calling Lock or another function that returns a lock. It has the same effect as the AmigaDOS CD command, changing your working directory. The following are equivalent calling sequences:

```
success = Execute("cd df0:c",0,0);
```

and

```
mylock = Lock("df0:c",ACCESS_READ);
oldlock = CurrentDir(mylock);
```

The advantage of the second form is that there need be no RUN command in the C directory of the currently assigned disk.

Only locks obtained from the Lock function should be unlocked later on. *Do not* call UnLock for a value that you obtain from the CurrentDir function. You see, AmigaDOS creates its own locks for dealing with the current directory; the value it returns is simply a pointer to AmigaDOS's own private lock. If your program unlocks it, AmigaDOS will not be able to access that disk or directory. Thus, only call UnLock for a lock that you received through the use of the Lock function.

Getting Information about a File or Directory

You use the Examine function to get information about a file or a directory. A call to Examine takes the following form:

```
success = Examine(lock,address_of_FileInfoBlock);
```

The first parameter is a pointer to a lock, usually obtained from a Lock function call. The second parameter is the address of a FileInfoBlock. When you call the Examine function, a FileInfoBlock is filled with information about a directory. Among the items contained in this FileInfoBlock

are the DirectoryType, FileName, Protection bits, Size, Comment, and so on. The meanings of the fields in this FileInfoBlock are summarized in the example shown in Listing 2.7.

```
/* exam.example.c */

#include "libraries/dos.h"
#include "exec/memory.h"

long rmask = ((long)('r') << 24);
long brmask = ((long)(' ') << 24);  /* a blank in same position */
long wmask = ((long)('w') << 16);
long bwmask = ((long)(' ') << 16);
long emask = ((long)('e') << 8);
long bemask = ((long)(' ') << 8);
long dmask = (long)('d');
long bdmask = (long)(' ');

struct
{
    long pmask;
    char stringnull;
}
    maskout;          /* place to build the protection bits value */

main()
{
    struct FileInfoBlock *fib;
    int success, p;
    struct FileLock *lock;
    extern struct FileLock *Lock();
    /* let's examine the 'dir' command file */
    fib = (struct FileInfoBlock *)AllocMem(sizeof(struct
          FileInfoBlock),MEMF_CLEAR);
    lock = Lock("df0:c/dir", ACCESS_READ);
    if(lock)
    {

        success = Examine(lock,fib);
        if(success)
        {
            printf("\n File name: %ls",&fib->fib_FileName[0]);
            if(fib->fib_DirEntryType > 0)
                printf("\nis a directory");
            else
                printf("\nis a plain file");
                                /* now calculate the protection bits */
            p = fib->fib_Protection;
            maskout.pmask = 0;
            maskout.stringnull = '\0';  /* end of string null */

            if(p & FIBF_READ)
                maskout.pmask |= brmask;
            else
                maskout.pmask |= rmask;
            if(p & FIBF_WRITE)
                maskout.pmask |= bwmask;
            else
                maskout.pmask |= wmask;
            if(p & FIBF_EXECUTE)
                maskout.pmask |= bemask;
```

Listing 2.7: The exam.example program

```
            else
                maskout.pmask |= emask;
            if(p & FIBF_DELETE)
                maskout.pmask |= bdmask;
            else
                maskout.pmask |= dmask;

            printf("\nhas protection bits of: %ls",&maskout);
            printf("\nhas a file size (bytes) of %ld",fib->fib_Size);
            printf("\n(%ld blocks)",fib->fib_NumBlocks);
            printf("\nIts file comment:\n%s",fib->fib_Comment);

            /* There is also a datestamp in this FileInfoBlock */
            /* that can be interpreted by the ShowDate function */
            /* (see Finding the Current Date later in this chapter.) */
            printf("\nhas a last-modified-date of:");
            ShowDate(&(fib->fib_Date));
        }
    Unlock(lock);
}                       /* end of exam.example.c */
```

Listing 2.7: The exam.example program (continued)

When you are in the root directory of a disk, the FileName field contains the volume name of the disk itself. Thus, for any disk, when you are in its root, you can read the volume name and the creation timestamp. It is through the use of the timestamp that you can tell the difference between two disks that have the same volume name.

Listing 2.7 is a program that calls the Examine function. The program lists only those data fields that might be of interest to the programmer. Other data fields are for AmigaDOS internal use only.

The reason that the AllocMem function is used to create the FileInfoBlock is that AmigaDOS requires it to be longword aligned. AllocMem always aligns on a longword (actually double-longword) boundary when it allocates memory.

Getting Information about the Next File or Directory

You use the ExNext function to get information about files within the same level in a directory. A call to ExNext takes this form:

```
success = ExNext(lock,address_of_FileInfoBlock);
```

The parameters are the same as those for the Examine function. ExNext uses the lock pointer and the current contents of the FileInfoBlock to determine which is the next file (if any) in the same level in the directory tree. It then fills information in the FileInfoBlock relating to that next file or directory.

The value of success is returned as zero when there is no item to be examined next.

Moving Up in the Directory Tree

If you have a lock on a file or a directory, the ParentDir function returns a lock on the parent; that is, it returns the directory of which this locked item is a part. A call to ParentDir takes this form:

```
parentlock = ParentDir(lock);
```

You use this function to move up the directory tree towards the root. When you reach the root, there is no parent directory available so the function returns a value of zero.

Program Using Directory Functions

Listing 2.8 is a program that implements some of the functions of the command string

```
DIR opt a
```

As a reminder, this command string tells AmigaDOS to list all of the files in the current directory, all of the directories within this current directory, and all of the files and directories beneath them. The difference between this and the system version is that this program outputs the names as it finds them rather than in a sorted sequence.

You can run the opta program from any directory path and it will list all files contained in that directory as well as all files and directories beneath it.

Determining the Current Working Directory

You can find the current working directory by taking advantage of a special feature built into AmigaDOS. By using a null string with the Lock function, AmigaDOS returns a lock on the current directory. You can get information about this directory and you can move to the parent of this directory. (The following program gets the directory name and moves to its parent.) In fact, by moving from parent to parent, you will eventually climb the directory tree structure all the way to the root of the current file system.

After successfully obtaining a lock, you can get information about this directory by using the Info command, or you can simply save this lock information for later. You may want to move away from the current directory and later return for some reason or other.

Listing 2.9 is a program that reports information about the current directory. Notice that it performs an Unlock function before it exits. You must unlock anything that you lock so that AmigaDOS can keep things straight. This is akin to freeing any memory you allocate before your program is finished.

```
/* opta.c */

#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"
extern struct FileLock *Lock(),*DupLock(),*CurrentDir();
main(argc, argv)
        int argc;
        char *argv[];
{
        struct FileLock *oldlock;
        struct FileLock *newlock;

        char *whichdir;

        if(argc == 1)
        {
        /* no directory specified, use current one! */
                whichdir = "";
        }
        else
        {
                whichdir = argv[1];
        }
        newlock = Lock(whichdir,ACCESS_READ);
        if(newlock != 0)
        {
                oldlock = CurrentDir(oldlock);
                followthread(newlock,0);
                oldlock = CurrentDir(oldlock);
                UnLock(newlock);
        }
        else
        {
                printf("can't lock selected dir\n");
        }
                printf("\n");
}

/* now follow the thread... might hit a directory, might hit a file. */
/* If a directory, list it and then follow it down (recursively). */
/* If hit a file, list it and proceed to the end. */

int followthread(lock,tab_level)
struct FileLock *lock;
int tab_level;
{

    struct FileInfoBlock *m;
    struct FileLock *newlock,*oldlock,*ignoredlock;
    int success,i;

    /* if at the end of the road, don't print anything */

if(!lock) return(0);
/* allocate space for a FileInfoBlock */

m = (struct FileInfoBlock *)
        AllocMem(sizeof(struct FileInfoBlock),MEMF_CLEAR);

success = Examine(lock,m);
             /* The first call to Examine fills the FileInfoBlock */
             /* with information about the directory. If it is */
```

Listing 2.8: The opta program

```
                /* called at the root level, it contains the volume */
                /* name of the disk. Thus, this program is only */
                /* printing the output of ExNext rather than both */
                /* Examine and ExNext. If it printed both, then */
                /* it would list directory entries twice!! */

    while (success != 0)
    {
        if(m->fib_DirEntryType > 0)
        {
                /* since it is a directory, get a lock on it and */
                /* go into it to list its contents as well as the */
                /* name of the directory */

        newlock = Lock(&m->fib_FileName[0],ACCESS_READ);

        /* If lock is valid then make this directory the */
        /* current one, but save the old lock value so */
        /* that we can return here and continue to list */
        /* the rest of the files and directories located here. */

        oldlock = CurrentDir(newlock);
        /* move into that directory */

        /* recursively follow the thread down to the bottom */
        followthread(newlock,tab_level+1);

        /* after listing the contents of the new directory, */
        /* come back here */

        ignoredlock = CurrentDir(oldlock);   /* and proceed */
        }
        success = ExNext(lock, m);           /* examine the next entry */
        if(success)
        {
                printf("\n");
                for(i=0; i<tab_level; i++)
                printf("\t");
                /* tab in to show directory level */
                printf(&m->fib_FileName[0]);
                if(m->fib_DirEntryType > 0)
                {
                    printf(" [dir]");
                    /* tell user this is a directory */

                    }
            }
        }
        if(lock) UnLock(lock);
        FreeMem(m,sizeof(struct FileInfoBlock));
    }
```

Listing 2.8: The opta program (continued)

The mybranch program uses a recursive call to the function named followpath. Its job is to continue to call the ParentDir function until it obtains a lock value of zero and to print the name of the directory it pops into at each step. The program prints a colon when it reaches the root of the disk and a slash for each subdirectory along the way to the path from which the program was started. When followpath reaches the root directory, the lock that is returned gives you the volume name of the disk.

```
/* mybranch.c */

#include "libraries/dos.h"
#include "libraries/dosextens.h"
#include "exec/memory.h"
extern struct FileLock *Lock(),*DupLock(),*ParentDir();
main()
{
          struct FileLock *mylock,*oldlock;
          /* get a read lock on the current directory  */
          oldlock = Lock("",ACCESS_READ);

          if(oldlock != 0)
          {
                    printf("\nPath to the current directory is: ");

                    /* don't print a slash if at bottommost level */
                    followpath(oldlock,0);
          }
          else
       {
                    printf("\n Can't lock the current directory");
       }
       /* NOTE: in this example, followpath unlocks the lock */
}

int followpath(lock,printslash)
struct FileLock *lock;
int printslash;
{
       struct FileInfoBlock *myinfo;
       struct FileLock *newlock;
       int success,error;

       /* if it reaches the end of the road, don't print anything */
       if(!lock) return(0);

       myinfo = (struct FileInfoBlock *)AllocMem(sizeof(struct FileInfoBlock),MEMF_CLEAR);
       if(myinfo == 0)
       {
              printf("Ran out of memory\n");
              return(0);
       }
       /* see if this directory has a parent; if so, pass it on  */
       newlock = ParentDir(lock);
       error = IoErr();
       /* newlock might fail because of an I/O error or because */
       /* somebody took out the disk */

       if(newlock == 0 && error != 0)
                 printf("\n DISK I/O ERROR!  value = %ld\n",error);

       /* recursively call this same function to follow path up to the root */

       followpath(newlock,1);

       /* file in the FileInfoBlock so we can print the name of this node */

       success = Examine(lock,myinfo);
       if(success)
       {
              printf("%ls",myinfo -> fib FileName[0]);
              if(newlock == 0)
```

Listing 2.9: The mybranch program

```
                        printf(":");
        }
           else
        {
        /* print a slash only if parameter is not zero */

        if(printslash) printf("/");
                UnLock(lock);
        }
        if(myinfo)
                FreeMem(myinfo, sizeof(struct FileInfoBlock));
        return(1);
        printf("\n");
}
```

Listing 2.9: The mybranch program (continued)

To try this program, compile it and name the executable file mybranch, then copy it to df0:c. Next, execute the following command sequence from the CLI:

```
CD df0:
MAKEDIR stuff
CD stuff
MAKEDIR morestuff
CD morestuff
CD
```

The CLI will respond with

```
df0:stuff/morestuff
```

Now type the command

```
mybranch
```

The CLI will respond with

```
VOLUMENAME:stuff/morestuff
```

where VOLUMENAME will be the name of your startup CLI disk (possibly C-CLI if you have been using the instructions that came with Amiga C).

AmigaDOS Utilities

The following utilities are available as AmigaDOS functions that your program can call directly:

- Rename a file or directory (Rename)
- Delete a file or directory (Delete)

- Create a directory (CreateDir)
- Protect a file or directory (SetProtection)
- Establish a comment for a file or directory (SetComment)
- Find out the current date (DateStamp)
- Get information about a disk (Info)

There are other utility functions that work with the multiprocessing system. These are discussed in Chapter 3, along with the multitasking system calls and data structure discussions.

Utilities Normally Accessed from the CLI

Of the above utilities, the following are normally accessed through CLI functions: Rename (the RENAME function), Delete (the DELETE function), CreateDir (the MAKEDIR function), SetProtection (the PROTECT function), and SetComment (the FILENOTE function). In this section, three alternative methods of calling these utility functions are provided, including using the Execute command.

As a notational convention, to distinguish between lines that you type into a CLI and lines that you use in a program, the CLI commands appear in all uppercase letters. Throughout this section, if a line begins with a word in all uppercase letters, it is a CLI command; if a line begins with the returned value of success, it is a line that might appear in one of your programs.

Renaming a File

The Rename function takes this form:

```
success = Rename(oldnamepointer,newnamepointer);

int success;
char *oldnamepointer, *newnamepointer;
```

The parameters are pointers to strings representing the old and the new file names respectively. If the value of success is zero, you can look at the value in IoErr to see what went wrong. Note that you can use this function to move a file from one directory level to another as long as the file stays within the same volume. For example, to move a file named myfile from df0:stuff/morestuff to df0:stuff, you would specify the oldnamepointer as df0:stuff/morestuff/myfile and the newnamepointer as df0:stuff/myfile.

To rename a file from the CLI, you can type

```
RENAME from oldname to newname
```

or you can use Execute, as follows:

```
success = Execute("rename from oldname to newname",0,0);
```

Another alternative is

```
success = Rename("oldname","newname");
```

Deleting a File

The Delete function takes this form:

```
success = Delete(currentname);

int success;
char *currentname;
```

The currentname parameter is a pointer to a string that describes the path name of the file to be deleted. If it is a name alone, it refers to the current directory. If the value of success is zero, you can look at the value in IoErr to see what went wrong.

To delete a file from the CLI, you can type

```
DELETE currentname
```

or you can use Execute, as follows:

```
success = Execute("delete currentname",0,0);
```

Another alternative is

```
success = Delete("currentname");
```

Creating a Directory

The call to CreateDir takes this form:

```
lock = CreateDir(dirnamepointer);

struct FileLock *lock;
char *dirnamepointer;
```

To create a directory from the CLI, you can type

```
MAKEDIR newdirectory
```

or you can use the Execute function:

```
success = Execute("makedir newdirectory",0,0);
```

Another alternative is

```
lock = CreateDir("newdirectory");
```

Notice that the call to CreateDir returns a lock on this newly created directory. If the lock value is zero, IoErr has information on why the function failed.

Protecting a File

The SetProtection function lets you specify a protection mask for a specified file name. The call to SetProtection takes this form:

```
success = SetProtection(namepointer,mask);

int success;
char *namepointer;
int mask;
```

Only the lowest four bits of the mask are significant. The significance is in the sequence RWED, standing for Read, Write, Execute, and Delete. If you set any of these bits, you are telling AmigaDOS that a file should be protected from being read; or protected from being written; or protected from being executed (as in the case of script files or normally executable binary files); or protected from being deleted. The state of the mask is just the opposite of what you normally see when you perform an AmigaDOS LIST command in that when the LIST command shows RWED as the protection flags, it means that the file *can* be read, written, and so on.

When you set one of these flags, it means that the corresponding bit will become set and thereby protect the file. AmigaDOS pays attention only to the D flag as of this writing. If a programmer creates a shell program (a program that looks and acts like an enhanced CLI), that shell program can use these other flags. For example, if the E flag is not set, the shell program might not try to execute that file.

To protect a file from the CLI, you can type

```
PROTECT filename DW
```

or you can use the Execute function, as follows:

```
success = Execute("protect filename dw",0,0);
```

Alternatively, you can use the SetProtection function:

```
/* ..... RWED ..... */
/* binary 0101 as W and D flags, */
/* write-protect and delete-protect */

success = SetProtection("filename",5);
```

Establishing a Filenote

You can use the SetComment function to add a filenote to a file or directory. A call to SetComment takes this form:

```
success = SetComment(namepointer,comment);

int success;
char *namepointer;
char *comment;
```

You can set a filenote from the CLI by typing

```
FILENOTE filename "This is a note I wanted to attach"
```

Or, in your program, you can use the Execute function:

```
success = Execute("filenote filename " "This is a note" " ",0,0);
```

Alternatively, you can use the SetComment function directly:

```
success = SetComment("filename","This is a note");
```

Finding the Current Date

You can ask AmigaDOS to provide you with the current date as it knows it, which is not necessarily accurate, unless accurate information is provided by the user. The function is called DateStamp, and a call takes the form

```
DateStamp(v);
```

where v is the address of the first of three longwords (32 bits each) that the function fills with the current date and time information.

You can convert the value received from the DateStamp function to a printable month, day, and year by the procedure in Listing 2.10. The program was written by Thomas Rokicki and is included here with his permission. Thanks, Tom, for this subroutine.

Getting Information about a Disk

After you have obtained a lock on a file or directory, you can use the Info function to obtain information about the disk on which this file resides. You pass a lock and an empty InfoData structure, and Info fills in the information about the disk. This is unlike the system INFO command, in that the Info function obtains information about one disk at a time, whereas the CLI command gathers and formats data about all block-structured devices in the file system at one time.

A call to Info takes this form:

```
success = Info(lock,address_of_InfoData);
```

```
char *months[]={"","January","February","March","April","May","June",
"July","August","September","October","November","December"} ;
long n ;
int m, d, y ;
main ()
{
            long v[3] ;
                                DateStamp(v) ;
        ShowDate(v);
}

ShowDate(v)
        long *v;
{
            n = v[0] - 2251 ;
            y = (4 * n + 3) / 1461 ;
            n -= 1461 * y / 4 ;
            y += 1984 ;
            m = (5 * n + 2) / 153 ;
            d = n - (153 * m + 2) / 5 + 1 ;
            m += 3 ;
            if (m > 12)
            {
                        y++ ;
                        m -= 12 ;
            }
            printf("%s %d, %d\n", months[m], d, y) ;
        return(0);
}
```

Listing 2.10: The printdate program

As with the FileInfoBlock, this InfoData structure must be longword aligned.

The info.example program in Listing 2.11 uses the Info function to get, then report information about whichever disk has the DIR function on it. This will normally be your Workbench or CLI disk.

Miscellaneous Functions

The functions discussed in this section are not available directly as CLI commands, though the information that they convey is often available from other commands, such as LIST. Functions are included here to do the following:

- Discover the name of the volume from which you booted
- Delay for a specific period of time
- Determine the process that is handling an I/O device
- Change the name of a disk volume

Discovering the Name of the Boot Volume

You can find out which volume name was used as the original Workbench boot disk (the disk you inserted just after Kickstart, when the

```
/* info.example.c */

#include "libraries/dos.h"
#include "exec/memory.h"

main()
{
    struct InfoData *id;
    int success, p;
    struct Lock *lock;
    struct
    {
        long pmask;
        char stringnull;
    }
    maskout;
    maskout.stringnull = '\0';

    /* let's get info about the disk where 'dir' is located */
        id = (struct InfoData *)AllocMem(sizeof(struct InfoData), MEMF_CLEAR);
    lock = Lock("df0:c/dir", ACCESS_READ);
    if(lock)
    {
        success = Info(lock, id);
        if(success)
        {
          if(id->id_DiskType == -1)
          {
             printf("\nNO DISK PRESENT");
          }
          else
          {
            printf("\nSoft Errors So Far: %ld",id->id_NumSoftErrors);
            printf("\nUnit # where (is/was) mounted: %ld",id->id_UnitNumber);
            printf("\nDisk State: ");
            if(id->id_DiskState == ID_WRITE_PROTECTED)
                printf("Write-Protected");
            else if(id->id_DiskState == ID_VALIDATED)
                printf("Read/Write");
            else if(id->id_DiskState == ID_VALIDATING)
                printf("Validating Disk File Structure");
            printf("\nDisk has %ld blocks",id->id_NumBlocks);
            printf("\nof which %ld are in use",id->id_NumBlocksUsed);
            printf("\nThere are %ld bytes/block",id->id_BytesPerBlock);
            printf("\nDisk Type is: ");
            maskout.pmask = id->id_DiskType;
            printf(&maskout);
            if(id->id_InUse == 0)
                printf("\nDisk is not in use");
            else
                printf("\nDisk is in use");
          }

        }
    }
}               /* end of info.example.c */
```

Listing 2.11: The info.example program

system asked for a Workbench disk) by passing a zero value as the lock for the Examine command.

No matter how many disk swaps you perform, along with ASSIGN statements, the system still remembers which disk volume name was used for the original boot.

The bootname program shown in Listing 2.12 uses the Examine function to read and report the name of the disk from which you booted the Amiga.

Delaying for a Specific Period of Time

You can use the AmigaDOS Delay function if you wish to put your task to sleep for a period of time denominated in fiftieths of a second. The call takes this form:

```
Delay(time);

int time;
```

Here's an example:

```
Delay(150);                    /* put task to sleep for 3 seconds */
```

Determining the Process Handling a Particular I/O Device

This technique is necessary if you wish to use a couple of the more advanced functions of AmigaDOS. Instead of executing an AmigaDOS utility function as demonstrated above, for certain functions you must send a message packet to a file process. This and the next section complete the AmigaDOS topics, but message passing is covered in the next chapter, so an explanation of the message-passing technique can be found there.

```
/* bootname.c */

#include <libraries/dos.h>
#include <libraries/dosextens.h>
#include "exec/memory.h"

main()
{
        struct FileInfoBlock *myinfo;
        int success;

        myinfo = (struct FileInfoBlock *)
                AllocMem(sizeof(struct FileInfoBlock),MEMF_CLEAR);
        success = Examine(0,myinfo);
        if(success)
                printf("%ls\n",&(myinfo -> fib_Filename[0]));
        else
                printf("no success\n");
        FreeMem(myinfo,sizeof(struct FileInfoBlock));
}
```

Listing 2.12: The bootname program

For a particular AmigaDOS device, you determine which process is actually handling its I/O by using the DeviceProc function. A call to DeviceProc takes the form

```
proc = DeviceProc(name);
```

where name is a pointer to a null-terminated string containing the name of the device of interest. The name parameter could be "df0:", "df1:", or "" (the null string) if it should work on whatever the current directory is set to.

Renaming a Disk Volume

If you need to change a disk volume label from a program, you can use either the Execute function to call the RELABEL command, or you can send a message packet to the process that is handling the device containing that disk. The type of message packet is ACTION_RENAME_DISK. You find the process by using DeviceProc. An example program with comments is shown below for a simple case:

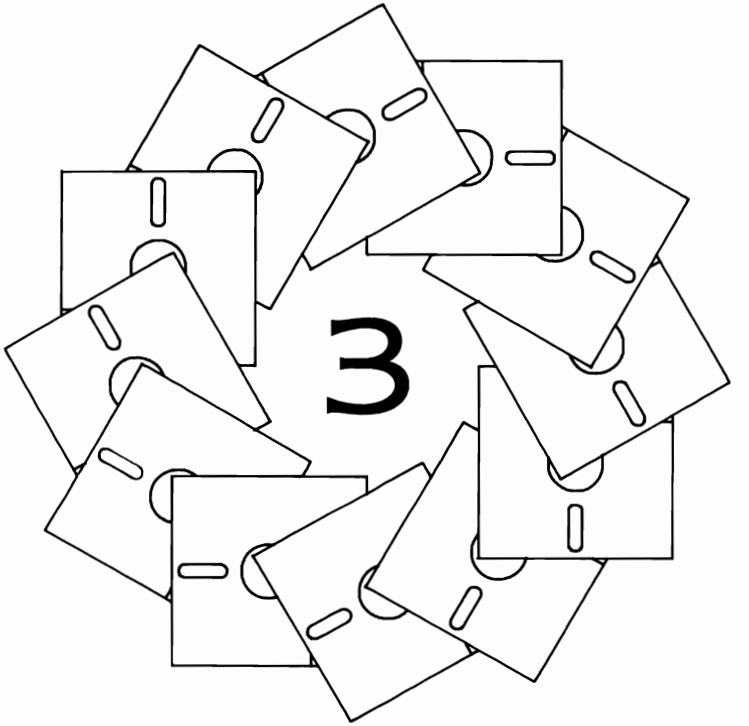
```
/* relabeldisk.c */

/* note: simply shows how to call Execute for this */
/*       function... programmer should build */
/*       own string for the RELABEL function **** */

main()
{
    int success;
    success = Execute("RELABEL oldname: newname",0,0);
    if(!success)
    {
        printf("Execute failed to relabel the disk\n");
    }
}
```

This chapter has covered nearly everything that AmigaDOS can do, from console-based input and output to file manipulation. You've seen how to open and close files, get and input data using AmigaDOS, and how to move around in and manipulate the file system.

The approach has been on cause and effect, rather than on listing internal data structures used by AmigaDOS. If you need more information about data structures, see the *AmigaDOS Technical Reference Manual* and the libraries/dos.h and libraries/dosextens.h Include files. The Include files are listed in the *Amiga ROM Kernel Manual*.

If you are trying to use the information contained in this book to adapt programs from other systems to the Amiga, there is another major area that you need to understand—device I/O. You'll find some information on devices covered in general in the next chapter, with specific devices covered in Chapter 6.

Exec



This chapter covers things you'll need to know to interact correctly with Exec. The following topics are covered here:

- The structure of Exec
- Some of its basic routines
- Some of its support functions

You won't find an in-depth examination of all of the functions of Exec. Many of the functions that Exec performs can be considered somewhat advanced. Rather than getting bogged down with advanced details here, you will find explanations of topics that are the basis for understanding the rest of the system.

THE STRUCTURE OF EXEC

Exec is a list-based multitasking executive. Everything that Exec manages is on a list somewhere in the system. *Multitasking* is a term that refers to Exec's ability to load and run many programs, switching back and forth between them so quickly as to make it appear that all programs, also called *tasks*, are running at the same time.

Following is a summary of the functions of Exec:

- It allocates memory on demand for tasks from a list of free blocks of memory.
- It controls its task switching by managing lists of tasks that are running, ready to run, or waiting for events to occur before they will again be ready to run.
- It maintains a list of ports at which messages can be posted from one task to another. Within a message port, Exec maintains a linked list of messages that have arrived at that port.
- It maintains a list of libraries of routines that allow tasks to share common program code and a list of devices (also called device drivers) that tasks can use for system I/O.
- It maintains a list of interrupt servers to manage the various hardware and software interrupts generated by the 68000, the custom chips, and the system software.

Why Lists Are Important

Exec uses linked lists to enable the Amiga operating system to be configured dynamically and to have no arbitrary limitations. When the system first boots (initializes), the entire system is freshly created.

Some parts of the Amiga system software require that memory space be set aside for their internal use. If your Amiga has memory outside the range of the custom chips (some form of expansion memory), the system software can select and utilize that outside memory effectively, leaving more space in the lowest chip memory for the custom chips to use. This gives you more space for graphics, for sound, for more sprite data, and so on.

Some Exec Functions and Terminology

In this section you'll find an examination of some Exec functions that manipulate or control the following:

- Tasks and processes
- Memory allocation
- Lists
- Signals
- Message ports
- Messages
- Libraries
- Devices

Those users who are primarily interested in Amiga's graphics can go directly to the next chapter, where the graphics discussion begins. Be aware, however, that we use some of the routines introduced in this chapter without further explanation in the graphics examples.

Note also that some understanding of messages, ports, and signals is essential for you to use device I/O information effectively. Device I/O is introduced in this chapter. I/O for specific devices is covered in detail in Chapter 6.

TASKS AND PROCESSES

As mentioned above, Exec keeps a list of tasks that can be scheduled to run on the 68000. If a task is awaiting some form of input, it can use

various system functions to put itself to sleep so that other tasks can have a chance to run. In addition, every time a system interrupt occurs—such as a timer interrupt, a keystroke or serial character received, or the completion of a video screen—Exec looks at its list of tasks and grants the use of the microprocessor to the highest priority task that is ready to run. As a result of the evaluation of the task list, the current task may be temporarily suspended in favor of a task at a higher priority.

For each task, Exec maintains what is called a *task control block*. There is only one microprocessor in the system, to be shared by all tasks. So Exec uses the task control block to store the values of all of the machine registers when that task is not running (also called "asleep"). The process of saving the state of one task's registers, restoring another task's registers, and making the new task active is called *task switching*.

When a task again becomes active ("awakens"), Exec restores all of its machine registers and the task continues operation as though it had never lost the use of the processor.

A *process* is a superset of a task. A process control block as defined for the Amiga—the name of the data structure is "Process"—contains a task control block as well as several other data structures utilized by AmigaDOS. The primary difference between tasks and processes is in the type of system functions that each can perform. In particular, if you wish to create a program that runs independently of the program that created it, and you wish to perform any AmigaDOS I/O functions, you will want to create a process rather than a task. AmigaDOS uses information contained in the process control block for its I/O operations. This information is not available in the basic task control block. Chapter 9 covers the topic of tasks and processes in detail.

MEMORY ALLOCATION

Exec maintains a list of free-memory areas in the system. Your own task or process can ask for chunks of memory from Exec. When you are through using the memory, you can return it to Exec to be put back on the list of available memory.

Once Exec has allocated a block of memory to your task, it no longer knows anything about that memory. Exec manages only free-memory areas, not allocated memory, so you must return memory to the system when you are finished using it. Otherwise it will be lost until the next system reboot.

Simple Memory Allocation

There are several levels of memory allocation that Exec can perform. We'll cover only the most basic method in this chapter; this is the method that is used most often in the examples in this book.

A call to the most basic memory allocation routine takes the form

```
address = AllocMem(size,requirements);
```

where size is the number of bytes that you wish the system to allocate for your own use, requirements tells the system what kind of memory you must use and what to do with it before the allocation request is fulfilled, and address is the starting (lowest) address in the block of memory that the system allocates for you. If the system cannot allocate a block of memory of the size you have requested, the call to AllocMem returns a value of 0. You must check the return value to ensure that you do indeed have the use of a memory block.

Here's what the requirements values mean:

MEMF_CHIP	Give me memory only in the area that the special-purpose chips can access.
MEMF_FAST	Give me memory only in the area outside that which the special-purpose chips can access.
MEMF_CLEAR	Set the contents of my memory block to all zeros before telling me where the block is located.
MEMF_PUBLIC	In preparation for possible memory management (virtual memory implementation), make sure this memory is allocated in a public (nonswappable) space, continuously accessible by all tasks.

When memory is allocated in MEMF_CHIP space, it means that the special-purpose hardware can get to the data that you write there. You'll use this requirement to provide display spaces, disk buffers, audio waveforms, and sprites.

When memory is allocated in MEMF_FAST space, it resides outside the range that the special-purpose chips can access. This memory is suited well for program space and nonDMA data space. Under some circumstances, such as heavy system DMA activity (high resolution, 16-color graphics or heavy data moving by the special-purpose chip called the Blitter), the special-purpose chip DMA activity can slow down the 68000.

If, while this heavy activity is happening on the special-purpose chip memory bus, the 68000 happens to be running a program from the extension RAM area, there will not be any bus contention and the 68000 will not slow down. Thus the name MEMF_FAST, since this space does not share memory cycles with the special-purpose hardware and can speed up the system.

Using MEMF_CLEAR, you can have an entire area preset to zero automatically. This is often convenient for initializing arrays or data structures and may save you some program code in the process.

The MEMF_PUBLIC requirement is not active as of this writing but is useful to add to a program's code for possible later releases of the operating system. This requirement will be applied to memory shared with the system DMA, assigned for use as a message or a message port between cooperating tasks, or assigned to interrupt program code and its associated data.

Returning Allocated Memory to the Free-Memory Pool

When you've finished using memory, return it to the system by using the FreeMem function. The call to FreeMem takes the form

```
FreeMem(address,size);
```

where address is the address of the memory block that you obtained from the call to AllocMem, and size is the size of the memory request that you placed when you called AllocMem.

In both the call to AllocMem and the call to FreeMem, the system automatically rounds the size of the allocation and frees up to the next multiple of MEM_BLOCKSIZE. (The current value of MEM_BLOCKSIZE is described in the Include file exec/memory.h.)

Program for Memory Allocation

The following program shows the correct syntax for calling the AllocMem and FreeMem functions. It also demonstrates that you should free the same amount of memory that you have allocated.

```
#include "exec/memory.h"
main( )
{
   char *address;

   address = AllocMem(300, MEMF_CHIP|MEMF_CLEAR);
   FreeMem(address,300);
```

```
    address = AllocMem(120, MEMF_FAST|MEMF_CHIP|
MEMF_PUBLIC);
    FreeMem(address,120);

    address = AllocMem(12345,0);
    FreeMem(address,12345);
}
```

You specify memory requirement combinations as a logical OR of the individual requirements. Specifying 0 as the requirement means "give me this size, in any type of memory available." The system defaults try MEMF_FAST first, then MEMF_CHIP.

LISTS

Lists are comprised of two basic data structures, the List structure, which is actually a list header, and the Node structure (also called a list node), which is a component of a list.

A list header can be thought of as the anchor of a system list. In fact, the basic routines for list manipulation always specify the list header as one of the parameters for the function call. In other words, the list header specifies which list you'll be manipulating.

Initializing a List Header

The important point about list headers is that they must be properly initialized before they are used. Fortunately, the Amiga library contains a function that you can use to do it right—the NewList function. The call to NewList takes the form

```
NewList(address_of_listheader);
```

where address_of_listheader is a pointer to the first (lowest) address of a memory block that is to serve as a list header. You needn't do anything to initialize this memory ahead of time; NewList handles it all.

Significance of List Nodes

The list node is simply a chunk of memory that is part of many of the other system data structures for which lists are built. The easy part of list nodes is that they need little or no initialization in order to use the basic system list routines. You can call AddHead (to add an item to become the first item of a list) or AddTail (to add an item at the end) and several other functions, without even knowing what is going on within the list node itself.

Figure 3.1 is a diagram that shows a list header and a couple of list items, each of which contains a list node as part of its own structure. Notice that a set of pointers within both the list header and list nodes is used to link these items into a complete list. Exec maintains both a forward pointer (to the next list item) and a backward pointer (to the previous list item) to make it easy to search a list.

In addition to the items within the list node that are used to maintain linked lists, there are two other fields that are used often by the system routines: the name field and the priority field.

The name field can be used to point to a null-terminated string that is to be the name of that node. Some of the system routines let you search for things by name, such as tasks or ports (FindTask, FindPort).

The priority field can be used to indicate to some of the system routines the sequence in which items are to be added to an existing list. List nodes that have a higher priority are inserted in a list ahead of nodes that have a lower priority. Routines that typically use the priority field are AddTask and AddPort. Priority values range from −128 to +127. For most applications, you use a priority value of 0.

Routines that Manipulate Lists

Exec provides the routines in Table 3.1 to manipulate lists directly. The parameters to these routines are as follows:

- node is a pointer to a node
- list is a pointer to a list header

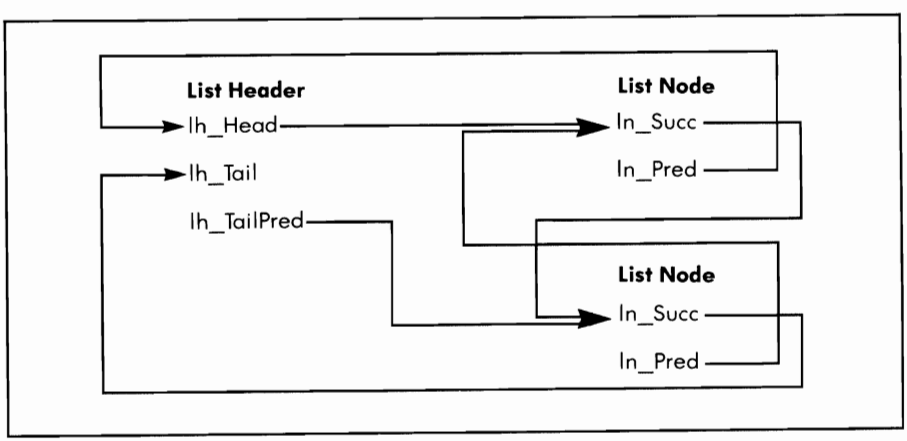


Figure 3.1: A linked list

- listnode is pointer to a node
- name is a pointer to a null-terminated string
- start is a pointer to a list header or a list node

The name search begins with the node *after* the one that start points to so as not to include the current node in the search; thus, the list header is a perfectly good place to start if you wish.

You can create and maintain your own special lists using these routines. Simply initialize a list header by calling the NewList function, create your own custom lists providing a list node as part of the data structure you wish to use, and manipulate the lists with the system routines.

Function	Purpose
AddHead(list,node);	Adds an item to the head of a list
AddTail(list,node);	Adds an item to the end (tail) of a list
Enqueue(list,node);	Adds an item to a list in priority sequence, ahead of the first item of a lower priority than this item
Insert(list,node, listnode);	Inserts a node on a list ahead of an existing node already in that list
Remove(node);	Unlinks a node from the list to which it is currently attached
node = RemHead(list);	Unlinks and returns the address of the node currently at the head of a list; returns a value of 0 if the list is empty
node = RemTail(list);	Unlinks and returns the address of the node currently at the tail of a list; returns a value of 0 if the list is empty
node = FindName (start,name);	Finds the next node in the current list that has the same name as that pointed to by the name field of the function call

Table 3.1: Functions for manipulating lists

Program Using List Functions

Listing 3.1 is a simple example that uses a list to implement a LIFO (Last-In-First-Out) queue.

It does not matter where in your own data structure the list node occurs as long as its starting address is fed to the list manipulation routines. For most of the data structures that the system uses, the list node is the first item in the data structure (e.g., struct MsgPort, struct Message, and struct Task).

However, for processes (struct Process), libraries (struct Library), and devices (struct Device), the list node is in a position in the data structure other than the lowest address. This happens because the rules under which these structures are used allow some kinds of data to be at positive offsets (higher addresses) and some kinds of data to be at negative offsets (lower addresses) compared to the position of the list node

```
/* list.example.c */

#include "exec/types.h"
#include "exec/lists.h"

struct MyListItem
{
        struct Node n;
        char *data;
};

main()
{
        struct MyListItem mli[3];
        struct MyListItem *mynode;
        struct List MyListHead;
        int i;

        NewList(&MyListHead);        /* init the list header */

        mli[0].data = "first ";
        mli[1].data = "second ";
        mli[2].data = "third ";

        for(i=0; i<3; i++)
        {
              AddTail(&MyListHead, &mli[i]);
        }

        for(i=0; i<3; i++)
        {
                mynode = (struct MyListItem *)RemTail(&MyListHead);
                printf("\n Just removed item whose data is: %ls",
                                        mynode->data);
        }
}                /* end of list example */
```

Listing 3.1: The list.example program

within the data structure. You will see this particularly in the discussion of libraries in this chapter.

SIGNALS

Signals are the means by which Exec controls what happens to a task as it is running. You tell Exec how your task is to react when a particular signal is received. The receipt of a signal is simply the setting (to a value of 1) of a particular bit within a longword (32 bits) in your task's task control block. You will not usually directly examine the signal bits within your task control block when you use signals. Rather, you will use the values returned by Exec to determine what has happened.

What Can Happen When a Signal Occurs

There are a few different things that can happen as a result of a task receiving a signal:

- Nothing happens if a task is not expecting to respond to this particular signal.
- A sleeping task awakens if it is inactive awaiting an event for which this signal bit was allocated.
- A task is forced into a special exception process if exception code and data has been provided and this signal, when received, is to cause an exception.

Note that exception processing is an advanced topic and is not covered in this book.

Significance of Signals in Multitasking

Signals are often sent to a task as a result of a message arriving at a message port that is owned by a task. There are 16 bits available to each task for user signalling and 16 bits assigned to Exec. If each message port owned by a task has a unique signal bit assigned, it is easy to determine which port has received a message by simply identifying the source of the signal. If you have a need to provide more than 16 message ports for a task, you can share a signal bit amongst several ports, then test each port, in turn, to see if there is a message present there.

Allocating a Signal Bit

Your task allocates a signal bit for its own use with the AllocSignal function. A call to AllocSignal takes the form

```
bitnumber = AllocSignal(number);
```

where number is a value from 16 to 31 if you want a specific signal bit number to be allocated, or −1 if you don't care about a specific number but simply want any signal bit that might still be available; and bitnumber is the value that Exec returns to you as the specific one of the signal bits that it allocates for your own use. If AllocSignal returns a value of −1, it was unable to fulfill your request. You must check the return value to see that a valid number has been returned. This value of bitnumber is the exact value that is needed in the data structure for message ports (mp_SignalBit) specifying which signal bit is to be set when a message arrives at a message port.

Here is an example of an AllocSignal call:

```
int signalbit;
signalbit = AllocSignal(-1);        /* give me any available signal bit */
if(signalbit == -1) error( );       /* do something if it went wrong */
```

Using Signal Bits in Multitasking

To promote efficiency in multitasking, Exec discourages busy-wait loops. For example, if you want to delay a function, you should set up a system timer somewhere, then put your task into a wait state until the timer causes a signal. If you are expecting a message to arrive from another task or an I/O operation to complete, you should not sit in a forever loop waiting for a bit to be set or a message to arrive. Such loops are useless time-wasters that might slow down the system unnecessarily, preventing other tasks with something useful to do from running.

The preferred method is to allocate a signal bit related to the item for which your task is to wait and to actually call the system Wait function specifying the bit or bits for which you wish to wait. For example, you might allocate one signal bit to signal arrival of a message; another signal bit may indicate a timer event has occurred, and so on. This frees the processor for other tasks. Your task will be made ready to run again when one or more of the bits you have requested becomes set. The system tells you which bits have become set as the return value from the Wait function, so you'll know what to do next. A call to Wait takes the form

```
wakeupmask = Wait(bitpattern);
```

where bitpattern is the logical OR of the signal bits for which your task is to wait. The occurrence of one or more of these bits as signals to your task will make it ready to run again. The returned value, wakeupmask, is the logical OR of all of the signals that have occurred to wake up the task.

Note that your task must respond to all of the bits contained in the wakeupmask, since this is the only time the bits will be reported as having been set. When your task wakes up, the bits that are reported as set in the wakeupmask are no longer set. Thus, if you are expecting two events to happen, each of which is associated with a unique signal bit, both events could occur within the same sleep period (depending on system tasking). Thus you must respond to all bits, lest you respond to one event and put the task to sleep again (another Wait) waiting for the other event that has already happened. This would cause your task to sleep forever.

Setting a Signal Bit Directly

There is another function that is used to set one or more signal bits in a task, but it is seldom used. Normally, signals are set automatically by such things as messages arriving at a message port. However, you could use a signal to allow one task to inform another that it has completed initializing a chunk of memory or perhaps has completed some custom arithmetic and has the result waiting in a commonly accessible memory area. The function is called SetSignal. The call to SetSignal takes the form

```
SetSignal(task,signalmask);
```

where task is a pointer to a task control block for the task that is to be signaled, and signalmask is a mask containing the bits that are to be set.

Using Multiple Signal Bits

Listing 3.2 is an example of waiting for multiple signals to occur. Notice that AllocSignal provides an integer value as a return. To convert that integer value to a physical bit number, you shift a 1-bit left by the number of positions in signalnumber. To wait for multiple signals to occur, you logically OR the signal bits into a single 32-bit mask. To test for particular signals, you logically AND the physical bit numbers against the wakeup mask. The listing assumes that you are waiting for either a keystroke or the receipt of a character from the serial port.

Notice that the code that tests the mask for each of the two signals independently should be positioned to ensure that if both signals were set, then both events would be noticed. Additionally, you must ensure that if multiple events occurred before your task was awakened, you

```
#define KEYSTRIKESIGNAL (1 << sigkey)
#define SERIALSIGNAL (1 << sigser)

int sigkey, sigser, wakeupmask;

sigkey = AllocSignal(-1);
if(sigkey == -1)
    error();

sigser = AllocSignal(-1);
if(sigser == -1)
    error();

/* Here might be code to initialize a message port and a Message */
/* data structure for communicating with the keyboard and assigning */
/* the signalkey to that port; and code for setting up */
/* a port and message for serial device communications, */
/* assigning the signalser to the port for serial communications */

/* Wait for either signal to occur by logically ORing the bits */

wakeupmask = Wait(KEYSTRIKESIGNAL | SERIALSIGNAL);

if(wakeupmask & KEYSTRIKESIGNAL)
{
/* code to process the key received */
}
if(wakeupmask & SERIALSIGNAL)
{
/* code to process the serial character received */
}
```

Listing 3.2: Waiting for multiple signals

process all of the things that could have caused that signal bit to be set. (Perhaps you must process all messages that have arrived at a message port because each arrival at a particular port sets the same bit.)

MESSAGE PORTS

A message port (MsgPort) is a data structure set up in memory as a place to which a task can send a message. A message is a block of memory that is located in one task's memory space. It contains information that some other task may need.

A port generally belongs to a particular task. When a message arrives at a port, several actions can be made to take place. The possible actions are indicated by the values for the port flags, called mp_Flags, in the MsgPort data structure:

PA_IGNORE	Do nothing when a message arrives
PA_SIGNAL	Signal the task that owns this port that a message has arrived

PA_SOFTINT	Cause a software interrupt of the task that owns this port

If the message is ignored, then it is simply added to the message list that the port maintains. Later, the task can perform the function GetMsg(msgport) to remove it.

If the message is to signal the task when a message arrives, then it is also possible that a task that has gone to sleep awaiting this message will then wake up and become ready to run.

If the message is to cause a software interrupt, it means that whatever the task was then doing will be suspended and a special set of interrupt code will be executed by that task. This could occur, for example, in a terminal program where it is important to retrieve every character that comes across the serial line even though the task is currently trying to finish processing of a keyboard keystroke.

Creating and Deleting a Message Port

The Amiga library function called CreatePort allocates memory for a message port and initializes various fields of the MsgPort data structure for you. The message port has a list header as part of the data structure, onto which lists of messages may be appended as a result of a call to the PutMsg function. CreatePort calls NewList to properly initialize this list header.

CreatePort allocates memory and a signal bit. If CreatePort has problems with either memory allocation (it can't find a large enough chunk of memory) or signal bit allocation (no more signal bits are available), it returns a value of 0. So you must check the return value to know that a port has indeed been created. You return the memory and deallocate the signal bit by calling the companion function DeletePort.

CreatePort initializes the mp_Flags to PA_SIGNAL and makes the calling task the owner of the port by initializing the mp_Task value to point to the calling task. Thus, the task that creates the port is the one that gets signaled when a message arrives there.

A call to CreatePort takes the form

```
mp = CreatePort(name,priority);
```

where name is a pointer to a null-terminated string that is to be the name of the message port and priority is the value to be applied to the priority field of the message port.

If the name field contains a nonzero value, this port is added to the system message-port list, using the AddPort function. This lets other tasks locate this port by name, using the function named FindPort. If the

name field contains a value of zero, AddPort is not called. Note that it is not necessary to add a port to the system if you do not need to rendezvous by name with the port.

If, for example, you have two ports with the same name, you can position one port ahead of the other in the system message-port list by setting the priority value of one higher than the other. When you use the FindPort function, even though both ports have the same name, the one with the higher priority will be found.

A call to DeletePort takes the form

```
DeletePort(mp);
```

where mp is a pointer to the message port that you obtained when you called CreatePort.

Since a port contains a linked list of messages, you should always be sure to get and reply to all messages that might be attached to this port before you delete it. Otherwise, you may be stopping another task that is waiting for your reply from completing its own activities.

Adding and Removing a Message Port

You add an initialized message port to the system message-port list by using the AddPort function. You remove a message port from the system with the RemPort function.

The call to AddPort takes the form

```
AddPort(mp);
```

The call to RemPort takes the form

```
RemPort(mp);
```

In both calls, mp is a pointer to a completely initialized message port. The main field that must be initialized is the priority field, so that Exec will know where to put the port in the system message-port list. You should also initialize the name field so that the message port will be accessible by name if desired. Also note that if the port is to be able to receive messages, the message list header must be properly initialized.

Notice that the CreatePort function calls AddPort for you if you have provided a name by which the port can be identified. CreatePort also appropriately initializes all of the other parts of the MsgPort data structure. DeletePort performs RemPort for you. Thus, by using these Exec support functions, you needn't even worry about AddPort and RemPort or any of the other initialization.

Finding a Message Port

You can locate a message port on the system message-port list by using the FindPort function. The call to FindPort takes the form

```
foundport = FindPort(pointer_to_namestring);
```

where pointer_to_namestring is a pointer to a null-terminated string of characters representing the name of the port to be found. A value of 0 is returned if a message port by that name is not currently on the system list of message ports.

Note that when a port is added to the system list, the priority field determines its location in the list. If there is more than one port on the system list having this particular name, the FindPort function finds only the port that has the highest priority in the ln_Pri field of the mp_Node field of the MsgPort data structure.

If your task suspects that there might be more than one port on the system list having the same name, you can use the FindName function to locate any lower priority message ports on this same list. In this case, the call to FindName would be made as follows:

```
struct MsgPort *mp, *mp2;

mp = FindPort("myport");

if(mp)          /* if nonzero (and suspect more than one by this name) */
{
   mpmore = FindName(mp,"myport");

   if(mpmore) printf("Found another one with the same name.");
}
```

Code Fragments for Using Message Ports

Listing 3.3 shows how a message port is allocated, how a signal mask is formed and used, and how a message port is deleted.

Watching for Signals from Message Ports

If a particular event is supposed to set a signal bit to inform a task that this event has happened, and the task does not immediately respond, it is possible that another such event will try again to set that same signal bit. Often this means, for example, that another message has arrived at a message port and will be appended to the list of messages that this port contains.

```
struct MsgPort *mp;                 /* pointer to a message port */
struct MsgPort *foundport;          /* another pointer */

int signalbit, signalmask;

mp = CreatePort("myport",0);        /* name it myport, at priority of 0 */

signalbit = mp->mp_SigBit;          /* find the signal bit it allocated */

signalmask = (1 << signalbit);      /* use bit number to form a waitmask */

/* now if we want to wait for a message */

wakeupmask = Wait(signalmask);

foundport = FindPort("myport");     /* shows how another task can locate */
                                    /* my message port so as to be able */
                                    /* to send messages to me. For this */
                                    /* code fragment, the value of foundport */
                                    /* should be the same as the value of mp */
if(foundport == 0)
{
        printf("can't find 'myport'");
}

/* and after everything is done, including responding to all messages */

DeletePort(mp);                     /* deallocates memory and signal bits as well */
```

Listing 3.3: Code fragments related to message ports

Your task cannot go to sleep waiting for a message to arrive at the port then simply process a single message before going to sleep again. Since there is only one signal bit that can be assigned to a message port for signaling the arrival of a message, it is not possible to have each message signal separately. Thus, a task must ensure that it responds to all messages appended to a message port after receiving a signal regarding that port, since messages already there can no longer set the signal bit to reawaken the task. In other words, messages queue, signal bits don't.

MESSAGES

A message is simply a block of memory that is used to pass data from one task to another. The block of memory belongs to the originating task.

The process of posting a message to a message port consists of linking the message onto the message list that is maintained by the message port. The Exec functions do not copy the message but simply modify a set of pointers to the block of memory so that the receiving task knows where to find the message information.

Why Use Messages?

Message passing is used extensively in the Amiga system to initiate I/O activity. For example, there are tasks that handle the serial and parallel ports, the keyboard, the gameport, and the disk. The process of initializing I/O on the system consists of formulating a message block that specifies the I/O activity that is to take place and passing that message block to the task that handles that specific hardware.

The receiving task is likely to be sleeping (letting your task run) while it waits for a message telling it what to do. When your task requests some I/O activity, your task may go to sleep waiting for the data to be returned. When the I/O task completes the request, it will return the message block to your task, indicating that the data is now available or signaling an error condition. Thus the I/O task may return to sleep, reactivating your task when the message is returned.

The Contents of a Message

The structure of a message is very simple:

```
struct Message
{
    struct Node mn_Node;                          /* a list node */
    struct MsgPort *mn_ReplyPort;     /* a pointer to a message port */
    int mn_Length;                     /* the length of the message */
};
```

The mn_Node is simply a list node, used to link a message onto a list. The mn_Node.ln_Type should be NT_MESSAGE, meaning "the type of this node is message." The mn_ReplyPort is a pointer to the message port to which this message is to be returned when the receiving task is finished using it. A message originates with one task, and is likely to be passed to another. This pointer to the reply port identifies the sender and tells to whom it is to be "replied" (returned). When the message is returned, the mn_Node.ln_Type is changed by the system to NT_REPLYMSG.

Often, the task sending a message creates a message port that will be used as the receiving place for messages, therefore designated as the sender's reply port. The sender may decide to send a message (using a function such as PutMsg), then go to sleep waiting for the message to be returned to its reply port.

The receiver task may have been asleep awaiting arrival of a message on its incoming message port. It may wake up, retrieve the message (using a function such as GetMsg), and return the message to the originator (using the ReplyMsg function) once it is finished using the contents of that message.

The length of the message, represented by mn_Length, is not checked by the system message-passing routines. You can assign your own significance to this value.

The Significance of Messages

Message passing is performed by reference, not by copying. When your task sends a message to another task, you are passing to it the address of a data structure within your own memory space. Your task is essentially giving temporary custody of this memory space to the other task.

The other task could copy the contents of the message into its own space (see Chapter 5), or write into this space (as is done with I/O requests), or whatever. When the other task has completed using this message, the message must be returned to the originating task by using the ReplyMsg function.

It is particularly significant that all messages be returned as quickly as possible. In particular, the task that sent the message may be in a wait state pending the return of the message to its reply port. Additionally, the originating task usually owns the memory space in which the message data is written.

Exec keeps track only of memory that is not allocated to any task, and each task can keep track of memory it has allocated and return it to Exec on exit. Thus, if a task never replies to the messages it receives, this noncooperating task may prevent other tasks from running or from returning memory resources to the system. Thus, messages sent with a reply expected must be "replied" in order to allow the system to continue to operate smoothly.

Your Own Custom Message

Certain of the system data structures are actually custom versions of messages that are used to pass information from a task to a specialized I/O task. The data structures named IORequest and IOStdReq are examples of this customization of the Message structure. A custom message structure can be set up as follows:

```
struct MyCustomMessage
{
   struct Message mcm_Message;
   int item1;
   int item2;
   int item3;
};
```

This shows a standard message structure with three integer items appended to it. This custom message can be manipulated using any of the system message-handling routines.

Functions that Handle Messages and Message Ports

Table 3.2 is a list of the functions that relate to messages and message ports. The parameters are as follows:

- msgport is a pointer to a message port
- msg is a pointer to a message
- priority is a byte value from −128 to +127
- name is a pointer to the first character of a null-terminated string

Program Using Messages and Message Ports

Listing 3.4 demonstrates message passing between ports. This would normally be done between tasks as a form of communications; however, this simple case is provided here for purposes of illustration. Chapter 9 demonstrates intertask message passing.

In this listing, the name field of the message node is used as the place in which the message was passed. Most intertask messages will likely not even be string-based, but will instead be a packet of information appended to the beginning or end of a Message data structure. Illustrating the general nature of these routines is what is important here.

LIBRARIES

A library is a group of related routines. When a library of routines is loaded into the Amiga, it is possible for all tasks to access the routines in the library. This can often make a program smaller because it need not include the code for many of the commonly required functions. For example, the Exec library contains the Exec-related routines such as list handling, task control, device I/O, message passing, and so on; the DOS library contains DOS-related functions; and the Graphics library contains graphics-related functions.

The Structure of a Library

A library is actually a data structure that can be linked into the system library list. This data structure contains a list node, a set of library

Function	Purpose
AddPort(msgport);	Adds a preinitialized message port to the system message-port list.
RemPort(msgport);	Removes a message port from the system list.
FindPort(name);	Finds the first port in the system message-port list of the specified name.
msg = WaitPort(msgport);	Puts a task to sleep awaiting the arrival of a message at a message port. Has a similar effect as the function Wait(1 << msgport->mp_SigBit). Points to the first message that has arrived at the port, but does not remove the message from the port. You still must use GetMsg to remove it.
PutMsg (msgport,msg);	Sends a message to a message port.
msg = GetMsg(msgport);	If there is a message present on that port, returns its address and delinks the message from the list of messages that the port is holding. If no message is present, returns a zero.
ReplyMsg(msg);	Replies to the message by transmitting the message (using PutMsg) to the message port specified in the ReplyPort field of the message itself.
msgport = CreatePort (name,priority);	Allocates memory for a message port and returns a pointer to it. If the name field is nonzero, adds this port to the system message- port list (using AddPort). Places the port into the list according to the value in the priority field. Thus it is possible to position this port ahead of a port that is already in the list. FindPort will find the highest priority port having a particular name.
DeletePort (msgport);	Deletes a port created by CreatePort. If the port has a name, it will have been on the system message-port list and is deleted from that list.

Table 3.2: Functions that handle messages and message ports

```
#include "exec/types.h"
#include "exec/ports.h"
main()
{
    struct Message m;              /* the message that we'll pass */
    struct Message *msg;           /* a pointer to a message that we'll retrieve */
    struct Message *GetMsg();
    struct MsgPort *mp;            /* a pointer to a message port */
    struct MsgPort *rp;            /* a pointer to another message port that is */
                                   /* to act as a reply port */
    struct MsgPort *CreatePort();
    extern

    mp = CreatePort(0,0);          /* Notice that it is NOT necessary to name */
                                   /* a port in order to be able to send a */
                                   /* message to it. It is only necessary */
                                   /* to name it if you wish to use FindPort later */
    if(mp == 0) exit(20);          /* error in CreatePort() */

    rp = CreatePort("reply",0);    /* name it reply, at priority of 0 */

    if(rp == 0)
    {
          DeletePort(mp); exit(30);
    }

    m.mn_Node.ln_Name = "Hello world\n";
    m.mn_ReplyPort = rp;           /* define the reply port */
    m.mn_Length = 0;               /* length is immaterial in this case */

    PutMsg(mp,&m);                 /* send the message to this port */
    WaitPort(mp);                  /* wait for it to arrive */

    /* since a message is already there, task does not even go to sleep */

    /* if we got here, we know there really is a message present. */

    while(msg = GetMsg(mp))
    {

    /* Get all messages from this port before either going */
    /* back to WaitPort or deleting the message port itself */

          printf("The message was: %ls\n", msg->mn_Node.ln_Name);
          ReplyMsg(msg);           /* send message back to its originator */
    }

    WaitPort(rp);                  /* Wait at the reply port for the return of the */
                                   /* original outgoing message... I won't have to call */
                                   /* ReplyMsg because I'm the one who sent it */
    while(msg = GetMsg(rp))
    {
          printf("Reply port received this message: %ls\n",msg->mn_Node.ln_Name);
    }
    DeletePort(mp);
    DeletePort(rp);
}
```

Listing 3.4: Message passing between ports

function vectors, and a data area. The structure of a library is illustrated in Figure 3.2.

The library node contains information about the library size as well as items that control library usage. Here is a list of the contents of the

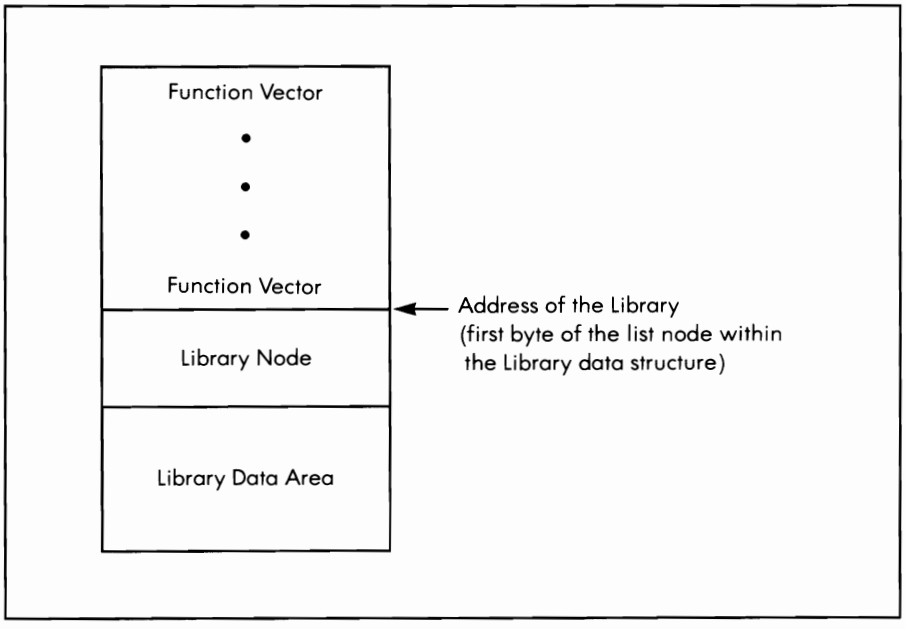


Figure 3.2: The structure of a library

library node:

```
struct Library
{
   struct Node lib_Node;
   UBYTE lib_Flags;
   UBYTE lib_pad;
   UWORD lib_NegSize;
   UWORD lib_PosSize;
   UWORD lib_Version;
   UWORD lib_Revision;
   APTR lib_IdString;
   ULONG lib_Sum;
   UWORD lib_OpenCnt;
};
```

It is important that you understand how the fields in the library node are used.

The lib_Flags Field

This field is used by Exec to keep track of what is happening to the library: whether someone changed a library vector; whether the library

is currently being checksummed; whether there is a delayed expunge pending. A programmer writing in assembly code that is intended to run as a result of the 68000 entering interrupt processing mode may have a need to examine bits within these flags, but the higher language programmer will likely not need to examine these bits at all.

Here is a code fragment showing what might happen with various flags in the library. Assume that the task that has just obtained control needs to use a routine contained in the library and wants to know if that routine is actually valid:

```
struct Library *lib;                          /* initialized elsewhere */

if(lib->lib_Flags & LIB_CHANGED)
{
                              /* maybe not consider valid if changed? */
}
if(lib->lib_Flags & LIB_SUMMING)
{
                        /* checksum not currently valid; library is not */
                      /* completely stable, may not wish to use it yet */
}
if(lib->lib_Flags & LIB_DELEXP)
{
          /* System is running low on memory and wants to dump this */
                /* library as soon as it can. When last user task closes */
/* the library it will go away. */
                                                                        }
if(lib->lib_Flags & LIB_SUMUSED)
{
                  /* info only... some task has set this flag to stop Exec */
                   /* from using the overhead that it takes to generate */
/* and/or check the checksum. */
}
```

The lib_Pad Field

This field is unused. Defining this field simply ensures alignment of the fields that follow it on word (16-bit) boundaries.

The lib_NegSize Field

This field tells how many bytes of data associated with the library precede the library node itself. This data area usually consists of a set of function vectors, each containing six bytes, each taking the assembly language form of a two-byte jump instruction followed by a four-byte address that is the target for the jump instruction.

The lib_PosSize Field

This field tells how many bytes of data associated with the library come after the library node itself. This is the library's data area that will usually be used to hold library global variables utilized by the library routines, or perhaps may even contain the library functions themselves, to which the library vectors refer.

The lib_Version and lib_Revision Fields

These fields contain numbers that uniquely identify the current version and revision number of a library. When you ask to open a library for use, you may have a special version of a library on your disk in the LIB: directory, and you may desire to use that library of routines in place of one that is already linked into the system by the Kickstart routines. By specifying a version number different from that already in the system, you can be assured of accessing your version of the routine from your own library.

The lib_IdString Field

This field contains the name of the library as it might be shown by a debugger (such as WACK). Libraries actually have two names associated with them: the name by which they are known to the OpenLibrary function (e.g., graphics.library, layers.library, or diskfont.library) and perhaps a longer name by which they can be identified more fully. This lib_IdString field may be as long as 255 characters if desired.

The lib_Sum Field

This is the checksum of the library function vectors. Once a library has been added to the system, Exec ensures the integrity of the library by performing a checksum on the function vectors. If a vector is changed, a new checksum is calculated. This checksum value is for Exec use only.

The lib_OpenCnt Field

This field contains the count of how many times the OpenLibrary function has been called for this library. When system memory is low, it is possible to automatically remove from the system any library that is no longer open to any user task. This returns this library's memory to the system for reuse. Every time OpenLibrary is called, this value is incremented. Every time CloseLibrary is called for this library, this value is decremented. When the value reaches 0, if Exec needs memory, the libraries with a lib_OpenCnt of 0 will remove (also called expunge) themselves from the system and return any allocated memory to the system free-memory pool.

Opening a Library

Before you can use the routines in a library, you must first open the library. The call to OpenLibrary takes the form

```
LIB_BASE = OpenLibrary(libraryname,version);
```

where version is the version number of the library you wish to use, LIB_BASE is a pointer variable having a name corresponding to the library you want to access, and libraryname is a pointer to a null-terminated string that names the library you want to open. It is possible that you wish to use a specific version of the library. In this case, specify that particular value. For example, version 31 is the number assigned for "V1.1" of the software system. If, on the other hand, you simply wish to use any version of this library name that is available, specify a value of 0.

OpenLibrary returns LIB_BASE—a pointer to the base address of a Library data structure. Note that this is a pointer to the first byte of a library node and that there will be portions of the library at higher memory addresses as well as portions of the library at lower memory addresses. If the OpenLibrary function is unable to fulfill your request, it returns a value of 0.

For each library, there is a specific name given for the variable that is named as the library base. Table 3.3 provides a list of the Amiga library names, base addresses, and types of routines contained in each type of library.

The library you wish to open and use may be either ROM/RAM resident or reside on disk. If it is not already in the memory system, AmigaDOS will search the directory currently assigned to LIB: to see if a library by that name is present there.

Library Base Addresses and Names

The significance of assigning a specific name to the library pointer you obtain from a call to OpenLibrary is that the Amiga library contains specific interface code for each routine in each library. This code is automatically linked into your program at link-time; it takes the value currently in that specific variable name and calculates, from that value, where it will find the jump vector for the routine itself. Then it takes the values that your C routine has placed on the stack, saves certain registers, loads your values into specific registers, then calls the specified routine.

This process of saving and restoring registers and calling the underlying routine is thoroughly explained in the *Amiga ROM Kernel Manual*. Suffice to say here that you must use the specific variable name and it must have a valid (nonzero) value before you call a function in that library.

Library Base Address		
Library Name	**Variable Name**	**Contents**
clist.library	ClistBase	Character string handling routines
diskfont.library	DiskfontBase	Disk-based font routines
exec.library	ExecBase	All Exec functions
dos.library	DosBase	DOS functions
graphics.library	GfxBase	Graphics functions
icon.library	IconBase	Workbench object functions
intuition.library	IntuitionBase	Intuition user interfacing functions
layers.library	LayersBase	Windowing/layering functions
mathffp.library	MathBase	Basic math functions
mathtrans.library	MathTransBase	Trancendental math functions
mathieeedoubbas.library	MathIeeeDoubBasBase	Double-precision IEEE math functions
timer.library	TimerBase	Timer arithmetic
translator.library	TranslatorBase	The Translate function

Table 3.3: Library names, base addresses, and contents

The Amiga operating system is dynamic in its design. Library code may, in fact, be loaded anywhere into memory either as the system is being built (during the Kickstart process) or by a call to OpenLibrary for a disk-resident segment of library code. The OpenLibrary function may load and initialize a disk-resident library and link it into the system library list for the calling task and other tasks to use. When a disk-resident library is loaded, its code, made relocatable by the linker, is scatter-loaded into memory chunk by chunk, with all function vector addresses

corrected to show exactly where the appropriate routine is now located.

Thus, once you have opened the library, the location of its library base will not change; you can store the library base variable and expect it to remain valid while you have the library open. Although the base address of a library will not change while it is resident in memory, it is possible that the function vectors within that library will change. Some programmers might be tempted, once they have found the library, to store the values found in the function vectors to speed up access to the function. Because the Amiga library provides a function that allows a task to change the values of a function vector within a library, it is always smartest to go through the standard library interface code so as to always execute the current version of the function in the library.

Using the Library Functions

Once a library is open, you can call any function within that library. This is simple from C language in that you need only specify the function name and its parameters:

```
result = Function(parameter1,parameter2);
```

The rest of the calling sequence to the routine is supplied by the Amiga library.

For assembly language programmers, there is a little bit more work to be done, although it is simplified by the assembly language macros that you'll find in a file named exec/libraries.i. First, you'll have to ensure that you've saved all of the appropriate registers on the stack and loaded all of the appropriate values into the registers that the routine will need. Then, if the library base value is already in register A6, use the CALL_LIB macro:

```
CALL_LIB _LVOFunction
```

If the value in A6 is not correct, use the LINK_LIB macro:

```
LINK_LIB _LibraryBase, _LVOFunction
```

Here, _LibraryBase contains the address of the base of the library in which the function vector resides, and _LVOFunction is the Library Vector Offset (LVO). This is the negative value that must be added to the library base address to create the address at which the function jump vector can be accessed.

Closing a Library

Once you have finished using the routines in a library, you should close the library to terminate your access to it. If you are the last user of a

library, it will be possible for the library to remove itself from the system and return the resources that it used.

You terminate access to a library by calling the CloseLibrary function. A call to CloseLibrary takes the form

```
CloseLibrary(libBase);
```

where libBase is the base address of that library.

Program to Open, Use, and Close a Library

Listing 3.5 opens a library, uses one of its functions, then closes the library.

Note that if you use the standard startup files from C—that is, Astartup.asm (produces Astartup.obj) or Lstartup.asm (produces Lstartup.obj)—you need not use OpenLibrary to get values for ExecBase or DosBase. The startup files do this for you. Therefore, in your program, you can feel free to call any routine in the Exec library or DOS library without opening or closing either of these two libraries.

```
#include "exec/types.h"
#include "intuition/intuition.h"
#include "intuition/intuitionbase.h"
extern struct Library *OpenLibrary();
LONG IntuitionBase *IntuitionBase;
main()
{

/* OPEN the library */

        IntuitionBase = (struct IntuitionBase*) OpenLibrary("intuition.library",0);
        if(IntuitionBase == 0)
        {
            printf("Open of intuition failed\n");
        }
        else
        {
            printf("intuition opened ok\n");

/* USE one of its routines */

            DisplayBeep(NULL);  /* flash the display */

/* CLOSE the library */

            CloseLibrary(IntuitionBase);
        }
}
```

Listing 3.5: Opening, using, and closing a library

DEVICES

A device, on the Amiga, is the name given to a data structure by which an I/O entity (such as the serial port, parallel port, console, and trackdisk) is accessed. Devices are constructed as a superset of libraries; in fact, they can be built by a system programmer using library code. There are routines for opening, closing, and expunging devices, just as there are similar routines for libraries.

As with libraries, devices are referenced by name, such as trackdisk.device, timer.device, and console.device. Again, as with libraries, a device may be memory resident or disk resident. When an OpenDevice function is called, if the device is not found on the system device list, Exec will search in the AmigaDOS directory currently assigned to DEVS: to see if there is a device with that name on disk. Exec then loads and initializes the device and adds it to the system device list.

Devices have units, which are instances of a device. For example, the TrackDisk device may have several units, each of which handles one of the physical disk drives attached to the system; the Timer device has two units, each a physical timer that it handles, one of which is more precise than the other. You'll find more about units under the heading Opening a Device later in this section.

In addition to the four standard function vectors common to all libraries (OPEN, CLOSE, EXPUNGE, and RESERVED), devices have two standard function vectors designated as the direct entry points to the device. These are the function vectors for BEGINIO and ABORTIO. Although these two entry points are available to you in the Amiga library functions BeginIO(iorequest) and AbortIO(iorequest), they are considered to be somewhat advanced and will not be covered in this book. Please see the *Amiga ROM Kernel Manual* for an explanation of these functions.

How I/O Is Requested

Device communications consists of formulating a message packet called an IORequest block (or I/O Request Block) and passing it to a message port owned by the device. Normally, there is an independent process started for each device. This process is sleeping, waiting for messages to arrive at its message port. The messages tell it what kind of I/O to do and how to do it.

The IORequest block contains a standard Message data structure that tells the process where to send the request block (to a reply port) when it is finished using it. It also identifies the type of command to be

performed as well as flags that tell how the command is to be performed and perhaps an address to or from which the data is to be transferred.

Device Commands

Device commands are used for I/O-directed activities. Table 3.4 lists the device commands and the values (given in io.h) that indicate the command number the device is to perform. This is the value that must be placed into the io_Command field of an IORequest block to ask the device to perform this particular command.

Opening a Device

You communicate with a device by passing an IORequest block to it. The actual passing of this message block is handled by a set of functions common to all device I/O, namely DoIO, SendIO, WaitIO, and CheckIO. These functions accept the message block from you and send it to the correct device by examining the io_Device and io_Unit data fields within the message. These fields are initialized by a call to OpenDevice.

You prepare a device for access by using the OpenDevice function. A call to OpenDevice takes the form

```
error = OpenDevice(DEVICENAME,unit,ioRequest,flags);
```

where error is equal to 0 if the call is successful, or nonzero if the device did not open.

DEVICENAME is the name of the device with which you wish to communicate. A list of the available device names that you can use in the OpenDevice call is given in a table in the next section.

The unit parameter is the identifier of the device unit that you wish to use. For the timer, for example, it might be UNIT_VBLANK or UNIT_MICROHZ. Each device has its own method of using this field. For most devices (Keyboard, Input, Console, Serial, and Parallel among others) this value can be zero.

The ioRequest parameter is a pointer to an IORequest data structure of a size appropriate to the particular device that you are trying to open. Exec uses information contained in your OpenDevice request to fill in portions of the uninitialized IORequest that you provide for further use. Essentially, OpenDevice is the initializer of a message block that, among other things, specifies the address of the device itself and uniquely identifies the unit of that device. For a Timer device, for example, you use a timerequest. For a Serial device, you use an IOExtSer. It is important that you pass the device an IORequest block of the

Device Command	Value	Purpose
Reset	CMD_RESET	Sets the device to its default values, restoring everything to the values the device had when it was first initialized
Read	CMD_READ	Reads a number of bytes into a specified memory area
Write	CMD_WRITE	Writes a number of bytes from a specified memory area
Update	CMD_UPDATE	Writes out the contents of internal buffers; if data is being cached before being written to disk, this command ensures that the disk will match the current contents of memory
Clear	CMD_CLEAR	Clears all internal buffers without updating; throws away the current contents of the buffer
Stop	CMD_STOP	Halts the device unit; allows the device to accept and enqueue new commands but prevents any queued commands from being executed
Start	CMD_START	Restarts a halted device unit
Flush	CMD_FLUSH	Aborts all I/O requests enqueued or in progress; all requests are returned with an error indicating I/O aborted

Table 3.4: Device commands and values

proper size since the device does initialize some parts of the request. If you pass it too small a request block, it is possible that the device will overwrite a different memory area, creating undesirable effects.

Some devices might have specific options that you wish to have enabled when the device is opened. The flags variable is reserved for that use.
The following code fragment illustrates opening the Timer device:

```
struct timerequest tr;
long error;

error = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);

if(error)
{
   printf("Timer open error; value %ld\n",error);
}
```

Names of the Commonly Available Devices

Table 3.5 lists the devices that are available either on the system as it boots up or can be loaded from disk and added to the system in response to the OpenDevice call. The Device Name column contains the actual null-terminated string (DEVICENAME) that you must put into the call to OpenDevice to access that particular device.

The Structure of a Typical IORequest Block

Here is a common form of an IORequest data structure; it is called a standard IO request, or IOStdReq:

```
struct IOStdReq
{
   struct Message io_Message;
   struct Device *io_Device;
   struct Unit *io_Unit;
   UWORD io_Command;
   UBYTE io_Flags;
   BYTE io_Error;
   ULONG io_Actual;
   ULONG io_Length;
   APTR io_Data;
   ULONG io_Offset;
};
```

An I/O request needs to have a method by which it can be linked into a list of pending I/O operations (io_Message) as well as a method by which the device and corresponding unit to operate on this request can be

Device Name	Purpose
audio.device	Controls the audio hardware channels
clipboard.device	Enables applications to pass data to and from a common area for such things as cut and paste between applications
console.device	Allows a window to perform user I/O as though the window were a terminal connected to a standard terminal-based computer
gameport.device	Controls the hardware of the gameports; both analog and digital inputs are accommodated
keyboard.device	Monitors and interprets keyboard input
input.device	Merges input from the Gameport and Keyboard devices into a time-synchronized stream of events to be interpreted by Intuition and possibly by a Console device
timer.device	Handles hardware timers for vertical blanking and a more precise timer for microsecond intervals
trackdisk.device	Handles the raw data to and from the disk
narrator.device	Handles requests for text-to-speech output
serial.device	Handles the serial interface hardware
parallel.device	Handles the parallel interface hardware
printer.device	Provides a common input format and a Preferences-selected variable output format for either serial or parallel printers that you might connect to the Amiga

Table 3.5: Commonly available devices

identified (io_Device, io_Unit). Both io_Device and io_Unit are supplied by Exec during the call to OpenDevice. These values should not be changed by the programmer.

An I/O request also needs to enable you to tell the device which command it is to perform (io_Command). Device-specific commands are enumerated in the Include files for each device. Common commands are enumerated in exec/io.h.

Formulating an I/O request often involves a data transfer of some kind, so the IOStdReq includes variables that specify:

- What special options to use if any (io_Flags)
- Where to find the data (io_Data)
- How many data bytes to send or receive (io_Length)
- At which position the data transfer should begin (io_Offset) if it is a block-structured device (such as a disk)
- Where the device should return the request after the I/O has been completed (io_Message.mn_ReplyPort)

The status information returned to the caller specifies:

- How many bytes of data actually got transferred as a return value from this request (io_Actual)
- If there was an error, what kind of error it was (io_Error)

Normally, io_Length and io_Actual will match. If they do not match, you may be able to find out why by examining the io_Error value.

Although IOStdReq is a very common data structure for passing information to and from devices, many of the Amiga devices have their own special version of an IORequest block structure. Each special form of request is discussed in the particular chapter covering that device.

Minimum Initialization Needed for an I/O Request

Although it varies depending on the device you wish to use, here is the barest minimum initialization that must be performed before any I/O request can be transmitted:

1. Open the device. This initializes the io_Device and io_Unit fields.
2. Initialize the following data fields of the request:

```
block.io_Message.mn_Node.ln_Type = NT_MESSAGE;
                                   /* a request block is a message */
```

```
block.io_Message.mn_Node.ln_Pri = 0;
                                        /* set the message priority */
block.io_Message.mn_Node.ln_Name = NULL;
                                          /* doesn't need a name */
block.io_Message.mn_ReplyPort = NULL;
                                    /* either NULL or an address */
                                      /* of a real reply port is OK */
```

(These are the data fields to initialize for an IOStdReq. You should initialize the corresponding fields of your own request block accordingly.)

3. Specify the command to be performed:

```
block.io_Command = WHATEVER;
```

Sending a Command to a Device

After opening a device, you fill in the appropriate fields in the IO-Request block to tell the device what you wish to do. Then you transmit this request to the device by one of several methods. These are the two most common methods of transmitting a request:

```
SendIO(request)          /* request I/O but don't wait till done */
DoIO(request)             /* request I/O and sleep until done */
```

About Request Blocks

Once you have transmitted a request block (or any message, in fact) to the device, even though this request block exists in the memory space belonging to your task, you should not read or write into that space until the message is returned to you at your reply port. This memory space is effectively given over to the device or other task for its own exclusive use. Only after it is returned to you should you attempt to reuse it.

Returning the request block to the reply port means that it is appended to the list of messages attached to that message port. To reuse it, you will have to call GetMsg(yourReplyPort) in order to unlink the message from the port.

About SendIO

SendIO is given a pointer to your request and attempts to append your message onto the device's message list. The device is managed by a separate task in the Amiga multitasking Exec. The request is performed in FIFO (First-In-First-Out) sequence. If the device is busy with a previous request, your task can go on to do something else and later check to see if the I/O has completed. Because your task does not wait

for the I/O to complete, this is known as an asynchronous I/O request.

When the I/O request has been fulfilled, whether successfully or unsuccessfully, your request block (message) is returned to your reply port. You must remove this message from your reply port before the request block can be used again. If you specify a reply port value of null, then no ReplyPort is used. You should use CheckIO to determine whether your I/O has completed.

About DoIO

DoIO is passed a pointer to your request. When you use DoIO to transmit the request, your own task is put to sleep until the requested I/O is completed or returned with an error. DoIO automatically removes your request block from the reply port before your task is awakened. If the reply port value is specified as null, the reply port is not used during the I/O.

To transmit the request to the device, there is a third method, the BeginIO(request) function. However, this requires specific knowledge of device internals, a topic not covered in depth in this book. Most examples in this book use either SendIO or DoIO.

Other I/O Functions

The other I/O functions in which you'll be interested are WaitIO, AbortIO, and CheckIO. The call to WaitIO takes the form

```
WaitIO(request);
```

If you have sent the request to the device using SendIO, and your program has reached a point at which you can go no further until the request has been completed, you use WaitIO to put your task to sleep pending the completion of the request. WaitIO automatically removes your request block from the reply port.

The call to AbortIO takes the form

```
AbortIO(request);
```

You use AbortIO to terminate a particular request, which may or may not have been executed already.

The call to CheckIO takes the form

```
result = CheckIO(request);
```

The returned result is a value of True if the I/O request has completed. You can decide to call AbortIO for this request if it has not yet completed.

Sample I/O Function Calls

The code fragments in Listing 3.6 are provided to show typical use of the I/O functions. (Additional examples can be found in Chapter 6.) These

fragments use the timer and the following data structures:

```
struct timerequest *message;          /* a pointer to a message */
                                      /* retrieved from a message port */
struct MsgPort myReplyPort;           /* the reply port at which the */
                                      /* message will be received when */
                                      /* the I/O request is returned */
                                      /* by the device */
struct timerequest myTimeReq;         /* a time request message block */
```

Each of the fragments assumes that the timer device has been opened using the following function call:

```
long error;

error = OpenDevice("timer.device",UNIT_VBLANK,&myTimeReq,0);
```

Each fragment also assumes that the timerequest is initialized by the following statements:

```
myTimeReq.tr_node.ln_Type = NT_MESSAGE;
                                      /* a request block is a msg */
myTimeReq.tr_node.ln_Pri = 0;;        /* set the message priority */
myTimeReq.tr_node.ln_Name = NULL;     /* doesn't need a name */

                                      /* and let's use a reply port */

myTimeReq.tr_node.mn_ReplyPort = &myReplyPort;
```

Finally, each fragment assumes that the time values have been initialized by the following statements:

```
myTimeReq.tr_time.tv_secs = 3;                /* 3 seconds */
myTimeReq.tr_time.tv_micro = 0;               /* 0 microseconds */
```

Notice that in code fragments 1 and 2, there was no need to remove the request block from the reply port, since DoIO and WaitIO perform this action for you.

Why Use a Reply Port?

As you have seen, it is not necessary to specify a reply port when arranging for device I/O in that the I/O can be completed acceptably by specifying a value of null instead of providing a reply port. Why, then,

```
/* Frag.1:  Put task to sleep awaiting I/O completion */

        DoIO(&myTimeReq);      /* wait for completion of request */

        /* now proceed to do something else... */

/* Frag.2:  Start the I/O, resynchronize with it later */

        SendIO(&myTimeReq);    /* start the timer */

        /* .... do a few other things .... */

        WaitIO(&myTimeReq);    /* go to sleep pending I/O completion */

        /* now go on to something else */

/* Frag.3:  Start the I/O, do some other things, check back */
            /* now and then to see if it has completed. */

        SendIO(&myTimeReq);

   otherthings:

        /* .... do a few other things .... */

        result = CheckIO(&myTimeReq);

        if(result == FALSE)
        {
            goto otherthings;
        }

        /* remove the request block from the reply port */

        message = GetMsg(&myReplyPort);

        /* and continue .... */

/* Frag.4:  Start the I/O, do some other things, check back */
            /* to see if it has completed. If not completed, */
            /* abort the attempt and continue. */

        SendIO(&myTimeReq);

        /* .... do a few other things .... */

        result = CheckIO(&myTimeReq);

        if(result == FALSE)
        {
            printf("Aborting the I/O request");
            AbortIO(&myTimeReq);
        }
        else
        {
            printf("The I/O completed as expected");
        }
        /* Remove the request block from the reply port */

        message = GetMsg(&myReplyPort);

        /* (value of message should be &myTimeReq) */

        /* and continue .... */
```

Listing 3.6: Code fragments using I/O functions

might a reply port be used? Well, recall earlier in this chapter where signals were discussed, you saw that a task can wait for one or more things to happen before it again becomes active. The receipt of a message at a reply port (message port) can be one of those events. Thus, setting up the reply port is a useful tool for multitasking, multievent sequencing.

Queueing Multiple Requests

If you want a device to perform several operations in a row asynchronous to your own task's operations, you can use SendIO to send multiple requests to the device. Notice that you must have a separate appropriately initialized request block for each of the requests. As noted earlier, the way to initialize a request block is usually by way of the OpenDevice function. However, if you are queueing multiple requests to the same device, then the io_Device and io_Unit values that you receive from the first OpenDevice function that you execute can be copied from the initial request block into each subsequent block, and all will be valid as long as your task has the device open.

Here is a code fragment that shows a second request being copied from the original one:

```
struct timerequest tr;
struct timerequest tr2;

error = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);
if(error == 0)
{
tr2.io_Device = tr.io_Device;
tr2.io_Unit = tr.io_Unit;
}
                        /* now either can be used for further device I/O */
```

Accessing a Device's Library Functions

The structure of a device and the structure of a library are extremely similar, and the device's function vectors are set up in the same way as those of a library. Certain devices, notably the Timer device and Console device, have functions that can be executed from C just as though they were library routines. These devices also have a library base address variable reserved. For the timer, the name applied is TimerBase. For the console, the name applied is ConsoleBase.

To use the routines in the device's library, you must provide the appropriate value for the base variables. You get the correct value by opening

the device and using the io_Device value that Exec returns to you in the IORequest block. Here is a code fragment that sets the timer variable correctly:

```
                                    /* Fragment ... setting the value for TimerBase */
                                      /* allows you to use time-arithmetic routines */
                                            /* AddTime, CmpTime, SubTime */

extern struct Library *TimerBase;
long error;
struct timerequest tr;

error = OpenDevice("timer.device",UNIT_VBLANK,&tr,0);

if(error == 0)
{
   TimerBase = tr.tr_node.io_Device;
}
else
{
   printf("error %ld while accessing timer"IOErr());
                                    /* (and do not try to use the timer functions) */
}
```

Once this value is established, you can use the time functions documented in the *Amiga ROM Kernel Manual*—AddTime, CmpTime, and SubTime.

The Console device has one routine in its device library, RawKeyConvert. The setting of the ConsoleBase variable is deferred to Chapter 5 because it requires a knowledge of windows and this concept has not yet been introduced.

Closing a Device

Once you have finished using a device, you must close it to terminate your task's access to it. This is especially important for disk-resident devices in that each device, after being loaded and initialized, takes up memory and perhaps processor time resources as long as at least one task has the device open. Thus, when you finish using a device, close it to allow the device to free the resources it is using and return them to Exec for reuse.

You close a device by calling the CloseDevice function. The call to CloseDevice takes the form

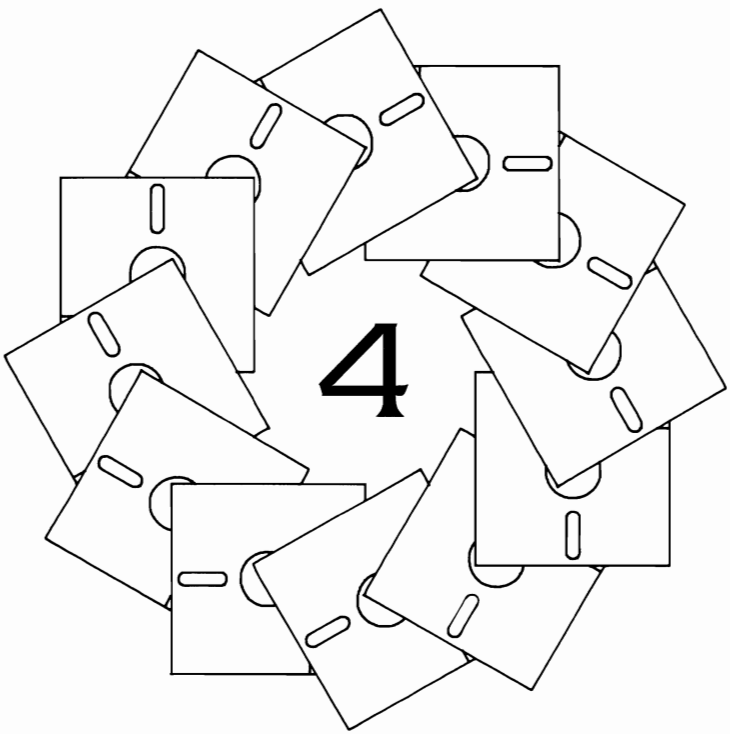
```
CloseDevice(request);
```

where request is a pointer to an IORequest block that you initialized by the OpenDevice function. Note that if you copied the original IORequest block to allow queueing of multiple requests, you must close the device using only one of the request blocks. In other words, call CloseDevice to close the device only as often as your task actually opened it.

In addition, before you close a device, make certain that the device has responded to all of the I/O requests that you have sent to it. If it has not responded to all of the requests, you should call AbortIO to abort those requests. When you do this, you are effectively emptying the device's list of messages to which it must respond for your own task before you tell the device that it is no longer needed by your task.

Devices can be shared among tasks. Each time a task opens a device, the device increments its user-count by one. Each time a task closes a device, the device decrements its user-count by one. When your task closes the device, if the user-count goes to zero, the device can remove itself from the system and free its resources.

Graphics



To uncover the unique graphics capabilities of the Amiga, this chapter provides useful tools that you will be able to incorporate into your own programs. All of the programs that you find here will be compatible with Intuition, the user interface of the Amiga. There are lower-level entry points to system routines, notably in graphics and layers, that will not be covered here in favor of creating programs that will be compatible with each other and with Intuition.

This chapter provides basic instructions on how to create and initialize drawing areas; how to specify and select colors; how to fill areas; and how to draw lines, circles, and boxes.

You will find applications here working first on the Workbench screen and then on a custom screen. You'll create a bar chart on the Workbench screen and a geographical map on a custom screen. Then you'll see how to program text for various fonts and sizes of type. Finally, you'll draw a picture.

OPENING A WINDOW ON THE WORKBENCH SCREEN

To create a window on the Workbench screen, you must specify the following:

- Where to put the window (where to put its upper-left corner)
- How large to make it (how wide, how tall)
- What title it should have in its title bar
- What colors to use to draw the border and system gadgets
- Flags to control the type of window you are creating and the system gadgets that are attached automatically to your window
- If your window can be resized, the minimum and maximum sizes within which you want it to be restricted
- The type of screen (in this case, Workbench) in which you want your window to open

Defining a New Window

To create a window, you define a data structure called a NewWindow. Listing 4.1 is a typical definition for a NewWindow. You will use this structure definition for the bar chart program.

```
/* window1.h */

struct NewWindow myWindow =
{
        30,                 /* LeftEdge for window measured in pixels, */
                            /* at the current horizontal resolution, */
                            /* from the leftmost edge of the screen */
        30,                 /* TopEdge for window is measured in lines */
                            /* from the top of the screen. */
        500, 150,           /* width, height of this window */
        -1,                 /* DetailPen - what pen number is to be */
                            /* used to draw the borders of the window */
        -1,                 /* BlockPen - what pen number is to be */
                            /* used to draw system-generated window */
                            /* gadgets */
                            /* (for DetailPen and BlockPen, the value */
                            /* of -1 uses default settings) */
        CLOSEWINDOW | NEWSIZE | REFRESHWINDOW;
                            /* IDCMP flags */
        SIMPLE_REFRESH | NORMALFLAGS | GIMMEZEROZERO;
                            /* window flags */
        NULL,               /* FirstGadget ... explained in Chapter 5 */
        NULL,               /* CheckMark ... explained in Chapter 5 */

        "Sample Chart",     /* window title */
        NULL,               /* pointer to screen if not Workbench */
        NULL,               /* pointer to bitmap if a superbitmap window */

        10, 10,             /* minimum width, minimum height */
        640, 200,           /* maximum width, maximum height */
        WBENCHSCREEN        /* type of screen in which to open */
};
```

Listing 4.1: The window1 structure definition

IDCMP Flags

Intuition is capable of sending messages to your task to tell you that a user has hit a particular kind of gadget or has done something you should know about. The messages that Intuition sends are received at the Intuition Direct Communications Message Port (IDCMP for short). In this program, you'll see a section in which the program waits for a message to arrive at the IDCMP. Then, based on the contents of that message, the program's next action is determined.

These are the flags that have been selected for this initial application:

- CLOSEWINDOW—so the application will know when the user wants to stop this program

- NEWSIZE—so that if the user makes the window larger or smaller, your application can redraw the window's graphics to occupy the blank space thus exposed

- REFRESHWINDOW—so that if the user pushes this window to the back or brings it to the front, the application can refresh (redraw) the window's contents

Window Flags

Three window flags are specified in the myWindow structure: SIMPLE_REFRESH, NORMALFLAGS, and GIMMEZEROZERO.

Setting the SIMPLE_REFRESH flag means that if a window is sent to the back, then brought to the front again, the system does not have to automatically save and restore the sections that were covered up. This saves memory, but the program must be capable of restoring the window's appearance after sensing that a refresh event has happened.

The NORMALFLAGS setting is something I've concocted for this application. Most often when I'm designing a display, I want my windows to have all of the usual system gadgets, as follows:

- WINDOWDRAGGING—lets you move the window around by using the mouse
- WINDOWSIZING—lets you resize the window
- WINDOWCLOSE—lets you signal the application that it should cease operation and close things down
- WINDOWDEPTH—lets you depth-arrange the window

NORMALFLAGS is thus defined by the following statements:

```
/* mydefines.h */

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAGGING

#define NORMALFLAGS (WC|WS|WDP|WDR)
```

When you open a window, a Window data structure is created and initialized. A RastPort data structure is a part of this Window structure. The rastport is used to control where and how drawing is performed by the graphics routines.

Setting the GIMMEZEROZERO flag means that the drawing area is created so that the coordinate location is in the upper-left corner of the area inside the window border (0,0). It also means that all drawing is clipped (cut off) within the borders of the window. If you don't specify GIMMEZEROZERO, the effective drawing area for a window is the

upper-left corner of the window itself and drawing can occur across the edges of the window border.

Opening the Window

Here is the code fragment that opens the window described by the myWindow structure:

```
struct Window *w;
w = OpenWindow(&myWindow);
if(w == 0)
{
   printf(''Window didn't open!'');
}
```

You provide a pointer to a NewWindow data structure (&myWindow), and the system returns a pointer to a Window data structure that it has initialized according to the parameters in that NewWindow structure.

If the return value is zero, it is possible that there is not enough free memory available for the system to use to create the Window data structure and all of its associated memory areas. If the returned value is nonzero, you know that you have a valid pointer to a Window data structure. It also means that the window you've described is now open on the Workbench!

Handling Events from Intuition

The IDCMP flags that you set in the NewWindow data structure tell Intuition to send three different kinds of events to the task. You'll need a routine that will do the appropriate thing if one or more of these events happens. This routine here will be called HandleEvent.

HandleEvent will be called from the main program, later in the chapter, to handle an event such as a mouse click on the CLOSEWINDOW gadget in your window, or a resizing or back-to-front request. If a CLOSE-WINDOW event is sensed, all HandleEvent needs to do is return a value of zero. This tells the main program that the program should stop. For a resizing or back-to-front request, the graphics should be redrawn, since this is a simple-refresh window.

Listing 4.2 is the code for handling these events. More event handling is shown in the next chapter.

Locating the Rastport

The next step you need to perform is to locate the rastport within this newly created Window data structure. A rastport is a data

```
/* event1.c */

HandleEvent(class)
     LONG class;          /* provided by main */
{
     switch(class)
     {
             case CLOSEWINDOW:
                     return(0);
                     break;
             case NEWSIZE:
             case REFRESHWINDOW:
                     redraw();
                     break;
             default:
                     break;
     }
               return(1);
}
```

Listing 4.2: The event1 routine

structure that controls drawing into a drawing area. For most of your programs, you need never know about the internal contents of a rastport. You simply need a pointer to a valid RastPort structure to pass to the system functions, which use the current contents of the rastport to determine how to fulfill your request.

Among the contents of a rastport are the current position of the drawing pen, the current drawing pen color, and the width and height of the currently selected text font. You can find an in-depth description of each of the RastPort structure variables in Chapter 2 of the *Amiga Programmer's Handbook*, vol. 1 by Eugene P. Mortimore (SYBEX, 1987). Structure definitions are beyond the scope of this book; structure variables will be discussed only as they are used.

You can locate the rastport for this newly created window by the following code:

```
struct RastPort *rp;                    /* a pointer to a rastport */
rp = &(w->RPort);                  /* a Window structure contains */
                                   /* a rastport, so we need to get */
                                     /* the address of that part of */
                                         /* the Window structure */
```

And that's all there is to it.

DRAWING INTO THE WINDOW

Thus far, the program segments have defined the characteristics of the window and have provided a pointer to the RastPort structure that

is to control drawing into the window. Now that you have completed the steps to make your program compatible with Intuition, you can go on to the code that actually uses the pointer to the rastport—the code that does the drawing. This code uses functions in the Graphics library.

In this section, you will create a bar chart. To make this exercise more useful, this application provides general-purpose tools for graphing.

Tools are provided here for the following:

- Selecting colors
- Selecting a drawing mode
- Drawing the axes
- Labeling the axes
- Drawing boxes
- Drawing dotted lines

Drawing a graph entails placing the axes somewhere within the window, then drawing the graph components. It is most convenient if you can treat the components of the graph relative to the graph's own coordinate system instead of the window's coordinate system. Thus, you need a set of routines that can do this.

The drawing functions shown here all use a common data structure to adjust for the graph's coordinates as compared to the window's coordinates. The system functions will be expecting window-relative X and Y, so these routines adjust to the system requirements. Listing 4.3 is the data structure that is used to hold the base values.

```
/* xybase.h */

struct XYBase
{
     WORD xaxis;          /* where x-axis is positioned in window */
     WORD yaxis;          /* where y-axis is positioned in window */
     WORD xlength;        /* how long to make the x-axis */
     WORD ylength;        /* how long to make the y-axis */
};
```

Listing 4.3: The XYBase structure definition

Selecting Colors

When you are working with the Workbench screen, there are four colors of drawing pens from which to choose a drawing color. The system sees these as pen numbers 0, 1, 2, and 3.

The colors that you'll be choosing for each pen can be viewed on the Preferences primary screen in the lower-left corner. Color 0 is leftmost, color 1 is next, and so on. For each pen number that you select for drawing, that particular color number will be used to render the image. Notice that pen number 0 is the background color.

There are three different kinds of drawing pens, known as the APen, the BPen, and the OPen.

The APen

The APen is the primary drawing pen color. When you are drawing a solid line, or a solid filled area, this is the pen number that is used. You establish the primary pen color by using the statement

```
SetAPen(rp,penNumber);
```

where rp is a pointer to the rastport to be affected, and penNumber is a value within the range of maximum colors available in the rastport. For this example, the maximum value is 3.

Later, if you wish to find out the value currently assigned to the APen, you can find it in the RastPort data structure, under the name FgPen (the foreground pen). You access this value, using a pointer to the rastport, as follows:

```
CurrentAPen = rp->FgPen;
```

The BPen

The BPen is the secondary drawing pen color. A patterned line or a patterned area can be formed by a combination of 1-bits and 0-bits in a pattern. When a patterned line or a patterned area is being drawn, the areas occupied with 1-bits in the pattern are filled with the primary (APen) color, and the areas occupied with 0-bits are filled with the secondary (BPen) color. This occurs only if the JAM2 drawing mode is set for this rastport. (A description of drawing modes follows.)

You establish the BPen color number by using the statement

```
SetBPen(rp,penNumber);
```

The BPen value is stored in the RastPort data structure as an item named BgPen (background pen). You can access this value, using a

pointer to the rastport, by the following code:

```
CurrentBPen = rp->BgPen;
```

The OPen

The OPen is also called the area-outline pen. If you are creating a filled polygon and if a value has been set for the OPen, then after the polygon is filled the system will automatically outline the polygon with a line of this color.

You establish the area-outline pen color number by the statement

```
SetOPen(rp,penNumber);
```

This value gets stored in the RastPort data structure in an item named AOlPen (area-outline pen). You can access this value, using a pointer to the rastport, as follows:

```
CurrentOPen = rp->AOlPen;
```

You may wish to retrieve this value and use it as your APen value when you are doing Flood operations. (Flood-filling is discussed in the custom screen example later in this chapter.)

Selecting a Drawing Mode

If you want to draw simple lines, select the drawing mode called JAM1. This means "jam one color into the drawing area." Let's draw the x- and y-axes for the bar chart in simple solid lines. You set the drawing mode by using the SetDrMd function, as follows:

```
SetDrMd(rp,JAM1);
```

JAM2 drawing mode means "draw using two colors, not just one." The APen color is used wherever there is a 1-bit present in the line pattern or area pattern. The BPen color is used wherever there is a 0-bit present in the pattern.

COMPLEMENT drawing mode means that whatever color is present in the target area, any 1-bit in the source area or source pattern causes the complement of the color in the target area to be drawn. A complement color is one in which the 1-bits become 0-bits and vice versa. For example, for a 4-bit color value 0101 (binary), the complement color is 1010. COMPLEMENT mode is useful when you wish to draw, then later erase, a line (such as producing a rubber-band line or a moving selector box). Draw the line once in COMPLEMENT mode; the line appears. Draw it a second time in COMPLEMENT mode; all of the bits in the line get restored to their original values and the line vanishes.

JAM1 and JAM2 modes can also be used in combination with INVERSEVID mode. This is explained in the discussion of text later in this chapter.

Drawing the Axes

Before you begin drawing, you have to position the drawing pen at the desired location, then draw from there to the end point of the line. This is accomplished by the Move(rp,x,y) and Draw(rp,x,y) functions.

The Move function picks up the drawing pen and puts it down in a new location. The Draw function draws a line, in the current drawing mode, from the current location to the location specified in the Draw function call.

Many different drawing functions can move the current position of the drawing pen. If, instead of specifically setting the pen position with the Move function, you wish to simply discover where the drawing pen is at the moment, you can examine the cp_x and cp_y variables, using a pointer to the rastport as follows:

```
CurrentXPenPosition = rp->cp_x;
CurrentYPenPosition = rp->cp_y;
```

Listing 4.4 draws axes according to the values in the XYBase structure that the DrawAxes function receives. Its inputs are a pointer to an XYBase structure that already shows where the axes are to be located, a pointer to the rastport into which the axes are to be drawn, the pen color in which the axes are to be drawn, and the axes placement and length values.

Note that for simplicity, no error checking is included. For example, if the value of yaxis minus ylength turned out to be negative, an error might result. Also note that DrawAxes is a programmer-defined function, not a function in the ROM kernel.

Labeling the Axes

To put labels on each axis, you need to know about using text on the Amiga. To use text, you must specify where in a rastport it is to be drawn, a pointer to the text that is to be output, and the length of the text string.

To specify where to put the text, use the Move function; it moves the drawing pen into position. When a text character is generated, the character begins as though the drawing pen were positioned at the baseline of the text character. For the standard system font, whose characters are all drawn within an 8-pixel-wide-by-8-line-tall rectangle, the baseline is the seventh line down from the top of the character's enclosing rectangle. You can discover the current baseline value by reading it from the TxBaseline value in your rastport. This can be done as follows:

```
CurrentTextBaseline = rp->TxBaseline;
```

```
/* drawaxes.c */

DrawAxes(rp,xyb,xaxis,yaxis,xlength,ylength,color)

        struct RastPort *rp;
        struct XYBase *xyb;
        LONG xaxis, yaxis;
        LONG xlength, ylength;
        LONG color;

{
        SetAPen(rp,color);

        /* draw the x axis */

        Move(rp,xaxis,yaxis);
        Draw(rp,xaxis + xlength,yaxis);

        /* draw the y axis */

        Move(rp,xaxis,yaxis);
        Draw(rp,xaxis,yaxis - ylength);

        /* now preset the values of the XYBase so that */
        /* other routines can use them. */

        xyb->xaxis = xaxis;
        xyb->yaxis = yaxis;
        xyb->xlength = xlength;
        xyb->ylength = ylength;
}
```

Listing 4.4: The drawaxes routine

When the character has been drawn, the drawing pen has automatically been moved one character position to the right, so as to start another character, at the baseline, if desired.

Once the drawing pen is positioned and the drawing modes and pen colors have been established, the function that you call to output the character is called Text. The call to Text takes the form

```
Text(rp,tptr,length);
```

where rp is a pointer to a rastport into which to render the text, tptr is a pointer to the text to output, and length is the number of characters to output.

To use Text for labeling axes, you have to position the text according to its length relative to the length of the axis. If the text is to be centered on its intended position, you need to know the length of the text to be output. For this, you can use the TextLength function. A call to TextLength takes the form

```
length = TextLength(rp,tptr,length);
```

where the values passed to the function are exactly the same as those for Text.

Listing 4.5 labels both the horizontal and vertical axes. The labels array is assumed to be an array of null-terminated strings. For convenience and speed, integer arithmetic is used.

It is more efficient to render text in long strings rather than by single characters for two reasons. First, there is overhead associated with the repeated subroutine calls. Second, if you have imposed any styling on the text, such as italics or boldface, the text will not be correctly rendered if you draw single characters. The underlying text output system can take a string of text and format it correctly as a string. For example, if a set of italic letters slightly overlap or lean into one another, a single call to Text will format a long string of text correctly. If, instead, you make a series of single character calls, you'll find that the characters are not rendered correctly.

Drawing Boxes

Bar charts are composed of boxes of various kinds. Some may be simple line drawings, some may be boxes filled with a color. The color may be either solid or patterned to further distinguish one bar from another on the chart. This section shows you how to use the system functions to generate boxes that are unfilled, filled, and patterned.

As with the other chart subroutines provided here, the boxes are generated relative to the chart coordinates rather than relative to the window coordinate system. Thus, a user can create the chart as though working on graph paper.

Unfilled Boxes

An unfilled box can be created by a Move function followed by four successive Draw functions. Since Move and Draw are shown in the preceding examples, the exercise is left to the reader.

Solid or Pattern-Filled Boxes

Filled boxes, either solid or patterned, can be produced by the system function called RectFill. A call to RectFill takes the form

```
RectFill(rp,xmin,ymin,xmax,ymax);
```

where rp is a pointer to the RastPort into which a filled rectangle is to be drawn; xmin,ymin is the upper-left corner of the rectangle; and xmax, ymax is the upper-right corner of the rectangle.

What happens as a result of the RectFill call depends on the current settings of certain rastport parameters, including the drawing mode,

```
/* labelaxes.c */

LabelHorizontal(rp,xyb,labels,howmany)

        struct RastPort *rp;
        struct XYBase *xyb;
        char *labels[];                 /* an array of label names */
        LONG howmany;                   /* how many labels are required */
{
        WORD i, labelwidth, segmentwidth, currentx;
        WORD actualy, actualx;

        segmentwidth = (xyb->xlength)/(howmany - 1);
        currentx = xyb->xaxis;

        actualy = xyb->yaxis + 1 + (rp->TxBaseline);
        /* put it below the axis line */

        for(i=0; i<howmany; i++)
        {
                labelwidth = TextLength(rp,labels[i],strlen(labels[i]));

                labelwidth = labelwidth/2;          /* center the text */

                actualx = currentx + (segmentwidth * i) - labelwidth;

                Move(rp,actualx,actualy);
                Text(rp,labels[i],strlen(labels[i]));
        }
}                 /* end of label horizontal */

LabelVertical(rp,xyb,labels,howmany)
        struct RastPort *rp;
        struct XYBase *xyb;
        char *labels[];                 /* an array of label names */
        LONG howmany;                   /* how many labels are required */
{
        WORD i, labelwidth, segmentheight, currentx;
        WORD actualy, actualx, currenty;

        segmentheight = (xyb->ylength)/(howmany - 1);

        currentx = xyb->xaxis - 2;  /* 2 pixels from axis */

        /* center the text vertically, using the text height */
        /* value contained in the RastPort */

        currenty = xyb->yaxis + ((rp->TxHeight)/2);

        for(i=0; i<howmany; i++)
        {
                labelwidth = TextLength(rp,labels[i],strlen(labels[i]));
                /* right-edge align the text along the axis */

                actualx = currentx - labelwidth;

                actualy = currenty - (segmentheight * i);

                Move(rp,actualx,actualy);
                Text(rp,labels[i],strlen(labels[i]));
        }
}                 /* end of label vertical */
```

Listing 4.5: The labelaxes routine

the area-fill pattern, and the primary, secondary, and area-outline pen colors.

For a solid color box, you can use the following code:

```
SetAPen(rp,mycolor);          /* preset the drawing pen color */
SetDrMd(rp,JAM1);                /* jam only one color into*/
                                          /* drawing area */
                                 /* then call RectFill */
```

For an outlined, solid-color box, use the above code, and prior to calling RectFill specify the area-outline pen color:

```
SetOPen(rp,myoutline);          /* preset the outline pen color */
```

Boxes with No Outline

Once the outline color has been established, the system will use that outline color thereafter for both RectFill calls and area-fill calls. (Area-filling is covered in the custom screen example later in this chapter.) To turn off the outline pen usage, you must use the system macro BNDRY_OFF:

```
BNDRY_OFF(rp);
```

Two-Color Pattern-Filled Boxes

To produce a pattern-filled box, you need to specify both a primary pen color and a secondary pen color, a drawing mode of JAM2, and a pattern that should be used for the fill. The pattern is an array of 16-bit words used by the area-filling functions as though they were stacked on top of one another to form a 16-bit-wide pattern in whatever size you specify. The main restriction is that the size of the box must be a power of two (2, 4, 8, etc.).

Listing 4.6 is a pattern that forms a checkerboard when displayed. When the graphics functions use this pattern, if JAM2 is the drawing mode, wherever there is a 1-bit in the pattern, the system draws with the APen drawing color. Wherever there is a 0-bit in the pattern, the system uses the BPen drawing color. If JAM1 is the drawing mode, the pattern is ignored and only the APen gets used.

To tell the system which pattern is to be used, use the SetAfPt function (set area-fill pattern). This function requires a pointer to the rastport that will be affected, a pointer to the pattern, and the size of the pattern you are providing (specified as a power of two). The sample pattern is 8 words long, so its size is 3 (2^3).

To draw a pattern-filled box, use the following code before calling RectFill:

```
SetAPen(rp,myPrimaryColor);
SetBPen(rp,mySecondaryColor);
```

```
/* mypattern.h */

UWORD mypattern[] =
{
     0xf0f0,              /* 4 bits of 1's and 4 bits of 0's */
     0xf0f0,
     0xf0f0,
     0xf0f0,
     0x0f0f,
     0x0f0f,
     0x0f0f,
     0x0f0f
};
```

Listing 4.6: The mypattern routine

```
SetAfPt(rp,mypattern,size);                    /* set pattern */
SetDrMd(rp,JAM2);              /* when drawing, use both pens */
                                              /* then call RectFill */
```

Multicolored Pattern-Filled Boxes

Instead of drawing a two-color pattern, you can draw a pattern having as many colors as are available in the drawing area itself. For this example, where the Workbench screen is used, there is a choice of four colors. In a custom screen, you can have a choice of up to 4,096 different colors for each bit position of a pattern (in hold-and-modify mode). See Appendix B of the *Amiga Programmer's Handbook*, vol. 1 (SYBEX, 1987) for a full discussion of the Amiga display modes.

The colors that are displayed on the Amiga screen are determined from a set of data bits that are taken from different parts of memory known as bitplanes. It is a combination of those bits that selects the color you see on the screen.

To display a multicolored pattern, you must specify the pattern in the form of a multilayered bitplane. It is the combination of the binary value of the bits taken from each bitplane of the pattern that determines the color displayed in a particular pattern position.

Here is how the bits are combined:

Bit in Plane 1		Bit in Plane 2		Pen Color Used
0	+	0	=	0
0	+	1	=	1

1 + 0 = 2

1 + 1 = 3

Here is a pattern that forms multicolored stripes:

Bit Position in Pattern Data

7 6 5 4 3 2 1 0

0 1 0 2 0 3 0 1

0 1 0 2 0 3 0 1

1 0 2 0 3 0 1 0

1 0 2 0 3 0 1 0

0 2 0 3 0 1 0 2

0 2 0 3 0 1 0 2

2 0 3 0 1 0 2 0

2 0 3 0 1 0 2 0

Listing 4.7 forms stripes on a background color.

You specify that the pattern is multicolored by the following code:

```
SetAPen(rp,255);                    /* draw into all available planes */
SetBPen(rp,0);                      /* BPen must be 0 */
SetDrMd(rp,JAM2);                   /* use JAM2 drawing mode */

SetAfPt(rp,mymulti, -3);  /* -3 represents the number of words that */
                          /* makes up each image of the pattern */
```

There must be as many images as there are bitplanes to be drawn into. Workbench is a four-color drawing screen; it thus requires two images. One image appears in one bitplane, the other image appears in the second bitplane. The value shown is −3 because there are 8 (2^3) words in each plane of the image.

About Patterns in General

Although the pattern capabilities of the Amiga graphics are impressive, there is one limitation: the patterns are always drawn with a starting point relative to the upper-left corner of the rastport drawing area. Thus, any shapes drawn with a pattern could appear as cutouts against a fixed-patterned background. This can be particularly disconcerting if

```
/* multipat.h */

UWORD mymulti[] =
{
        /* plane 0 part of the multicolored pattern */

        0x3033,
        0x3033,
        0xc0cc,
        0xc0cc,
        0x0330,
        0x0330,
        0x0cc0,
        0x0cc0,

        /* plane 1 part of the multicolored pattern */

        0x0330,
        0x0330,
        0x0cc0,
        0x0cc0,
        0x3003,
        0x3003,
        0xc00c,
        0xc00c
};
```

Listing 4.7: The multipat routine

you are trying to animate patterned objects by quickly drawing and redrawing them against a fixed nonpatterned background.

As an example, suppose you want to draw a pair of playing cards, each having the same pattern on the back, and slowly move one in front of the other. Based on the position of the moving playing card, you could change the pattern that the system uses for the area-fill so as to keep the appearance of the pattern the same no matter where on the screen the card moves. As an easier alternative, you can create an off-screen rastport, create your image there, then copy the image to the on-screen rastport, thereby keeping the pattern the same regardless of where the card is moved.

Adapting to the Coordinate System

Listing 4.8 adapts the rectangle coordinates from window-relative to axes-relative. It also includes specifying the position of the lower-left corner of the bar, and the width and height, translating this into what RectFill will understand. Note that the APen, BPen, drawing mode, and area-fill pattern must be properly set before calling the DrawBar function.

This could have gotten fancy and adapted to the bar as a percentage of the maximum X and Y, as was done with the axes labeling, but that's for you to work on if you wish.

```
/* drawbar.c */

DrawBar(rp,xyb,x,width,height)

        struct RastPort *rp;
        struct XYBase *xyb;
        LONG x, width, height;          /* where to put it, how big it is */
{
        WORD xmin, ymin, xmax, ymax;

        /* bar is to be drawn resting on the x axis */
        /* assumes that colors, outline, pattern and drawmode are OK */

        xmin = x + xyb->xaxis;          /* adjust for x axis position */
        xmax = xmin + width - 1;

        ymax = xyb->yaxis - 1;          /* rest it on the x axis */
        ymin = ymax - height + 1;

        RectFill(rp,xmin,ymin,xmax,ymax);
}
```

Listing 4.8: The drawbar routine

Dotted Lines

When the axes are drawn, solid lines are used. If you wish, you can apply a pattern to the lines, just as you are able to apply a pattern to the area-fills. A line pattern is established by the SetDrPt function (set drawing pattern).

The pattern itself is a 16-bit unsigned word, containing the set of 1- and 0-bits that define a patterned line. For example, a dotted line might appear as follows:

1100110011001100

When a line is drawn, if the drawing mode is JAM1, wherever there are ones in the pattern, the APen color will be drawn. Where there are zeros in the pattern, the background color or pattern, if any, remains undisturbed. If the drawing mode is JAM2, wherever there are ones in the pattern, the APen is used. Where there are zeros in the pattern, the BPen color is used.

To use the dotted-line pattern above, you would specify a line pattern value of 0xCCCC. If you do not explicitly select a line pattern, the default pattern is used. The default value is 0xffff, which creates a solid line.

Drawing Multiple Lines with a Single Function Call

In the main program listing that follows, dotted lines are used to connect the tops of two of the bar chart values and a solid line is used to

connect the third set. The drawing of the lines is performed by using a special function call that draws multiple connected lines with a single call. This function is PolyDraw and it takes the form

```
PolyDraw(rp,xytable,count);
```

where rp is a pointer to a rastport; xytable is a table of words, each pair of which defines an X,Y coordinate to connect in a set of interconnected lines; and count defines how many coordinate pairs are in the xytable.

The Main Program

Now that all of the pieces have been explained, Listing 4.9 is the main program body for the bar chart. To avoid repetition, the main program simply includes the pieces labeled individually earlier in the chapter.

Avoiding Redrawing Window Contents

The bar chart program uses a simple-refresh window for its drawing. This means that for any resizing operation, or any operation where another window has partially covered then exposed a portion of this window, it is necessary to use redraw to restore the window to its original appearance.

You can avoid the redraw operation: instead of a simple-refresh window, you can select either a smart-refresh or a superbitmap window.

Smart-Refresh Windows

When you select a smart-refresh window, the system automatically saves and restores any obscured then exposed part of your window when a back-to-front or window move operation happens. There are certain costs to this operation:

- Saving and restoring takes as much memory as is needed to save the obscured areas.
- When you draw into a smart-refresh window, if any part of the window is obscured, the graphics routines draw into all parts of the window so that when the obscured portion is exposed, the results of your drawing need not be redone. This is both an advantage and a disadvantage in that the more overlapped segments you have, the longer it takes the system to draw an image. But as fast as the Amiga draws, the time increase over a simple-refresh window may not even be noticed.

```
/* main barchart program */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/gfxmacros.h"

#define NORMALFLAGS (WINDOWSIZING|WINDOWDRAG|WINDOWCLOSE|WINDOWDEPTH)

#include "mypattern.h"
#include "multipat.h"
#include "xybase.h"
#include "window1.h"
#include "event1.c"
#include "drawaxes.c"
#include "drawbar.c"
#include "labelaxes.c"

struct Window *w;
struct RastPort *rport;

extern struct Window *OpenWindow();
int GfxBase;
int IntuitionBase;

main()
{
        struct IntuiMessage *msg;
        LONG result;

        GfxBase = OpenLibrary("graphics.library",0);
        if(GfxBase == 0)
        {
                printf("graphics.library won't open!\n");
                exit(10);
        }
        IntuitionBase = OpenLibrary("intuition.library",0);
        if(IntuitionBase == 0)
        {
                printf("intuition.library won't open!\n");
                exit(15);
        }

        w = OpenWindow(&myWindow);
        if(w == 0)
        {
                printf("Window didn't open!\n");
                CloseLibrary(GfxBase);
                exit(20);
        }
        rport = w->RPort;
        redraw();        /* (redraw for the "first" time) */

        /* Now wait for a message to arrive from Intuition */
        /* (task goes to sleep while waiting for the message) */

        WaitPort(w->UserPort);
        /* retrieve the message from the port */

        while(1)        /* "forever" */
        {
                msg = (struct IntuiMessage *)GetMsg(w->UserPort);
        handleit:
                result = HandleEvent(msg->Class);
```

Listing 4.9: The main barchart program

```
                if(result == 0)     /* got a CLOSEWINDOW */
                    break;
                /* It is possible that Intuition might send more */
                /* than one message to the task while the task */
                /* is waiting, so the event loop must empty the */
                /* port each time it goes to check its contents */

                msg = (struct IntuiMessage *)GetMsg(w->UserPort);

                if(msg != 0)        /* will be 0 when there are */
                                    /* no more messages */
                    goto handleit;
        }
        CloseWindow(w);
        CloseLibrary(GfxBase);
        CloseLibrary(IntuitionBase);

}
char *hlabels[] = {
                " ", "84", "85", "86", "87" };
char *vlabels[] = {
                "0", "10", "20", "30", "40" };

struct XYBase myxyb;
redraw()
{
        WORD i;

        DrawAxes(rport,&myxyb,35,120,350,100,1);

        LabelHorizontal(rport,&myxyb,hlabels,5);

        LabelVertical(rport,&myxyb,vlabels,5);

        SetAPen(rport,1);
        SetDrMd(rport,JAM1);

        for(i=1; i<5; i++)
        {
                DrawBar(rport,&myxyb,-31+i * 348/4,20,i * 12);
                /* where, offsets, xposition, width, height */
        }

        SetAPen(rport,1);
        SetBPen(rport,2);
        SetAfPt(rport,mypattern,3); /* 2 color pattern */
        SetOPen(rport,3);           /* outline it in third color */
        SetDrMd(rport,JAM2);

        for(i=1; i<5; i++)
        {
                 DrawBar(rport,&myxyb,-10+i * 348/4,20,i * 18);
                 /* where, offsets, xposition, width, height */

        }

        SetAPen(rport,255);
        SetBPen(rport,0);
        SetAfPt(rport,mymulti,-3); /* 2 color pattern */
        SetOPen(rport,1);          /* outline it in first color */
        SetDrMd(rport,JAM2);

        for(i=1; i<5; i++)
        {
```

Listing 4.9: The main barchart program (continued)

```
                DrawBar(rport,&myxyb,11+i * 348/4,20,i * 22);
                /* where, offsets, xposition, width, height */
        }
}               /* end of redraw */
```

Listing 4.9: The main barchart program (continued)

- If you resize the window to make it smaller, the system will save and restore only the portions of your drawing that exist within the final bounds of the window. When you resize the window to make it larger, the system can provide you with blank space only in the area surrounding the previous size. You see, when you draw into a smart-refresh window of a given size, any drawing that falls outside the boundaries of the window is clipped, that is, not drawn at all. Thus, a smart-refresh window, though it need not respond to REFRESHWINDOW Intuition events, may still have to respond to NEWSIZE Intuition events.

To use the smart-refresh feature, replace the SIMPLE_REFRESH flag in the NewWindow data structure with a SMART_REFRESH flag. Then you can eliminate the REFRESHWINDOW flag from the event-handling routine.

If you also wish to avoid redrawing the window contents on a NEWSIZE event, the alternative is simple: delete WINDOWSIZING from the window flags of the NewWindow data structure. If there is no sizing gadget present, the system will ignore the minimum and maximum sizing variables in the NewWindow data structure and not cause any resizing.

Superbitmap Windows

If you still want to be able to freely resize your window without having to respond to either the REFRESHWINDOW or NEWSIZE events, you may wish to use a superbitmap window instead.

When you select SUPER_BITMAP, you specify your own drawing area to link into the system display facilities. Any drawing that you do is always kept in your own drawing area. When a window section is obscured and then exposed, or when the window is sized up or down, the system pulls your drawing area into view. This means there will be no need to respond to either REFRESHWINDOW or NEWSIZE events.

This, too, has its costs:

- As with smart-refresh windows, this method takes more memory than simple-refresh windows. It also requires that you know about and provide a properly initialized bitmap and its associated memory.
- As with smart-refresh windows, this method adds a little to the drawing time in that some of the drawing appears on the screen and some of it appears off the screen.

An advantage to using a superbitmap window is that the bitmap can be as large as 1024-by-1024 pixels, and you can, if you wish, reposition your window on different portions of this larger bitmap regardless of the current size of the window. The default positioning as you first open the window is that the upper-left corners of the on-screen bitmap and the off-screen bitmap are aligned.

A superbitmap window must also be a GIMMEZEROZERO window. If it is not GIMMEZEROZERO, Intuition renders your gadgets and window borders into the superbitmap. If you later enlarge, shrink, or scroll the window against the superbitmap, the window borders and gadgets will still be there.

Listing 4.10 lists data declarations and a code segment that you can insert into the bar chart program just ahead of the call to OpenWindow to make this a superbitmap window. Note that you must put the data declarations into the declarations part of the main program.

First, you need to change the SIMPLE_REFRESH flag in the New-Window data structure to read SUPER_BITMAP. Also, you must remove NEWSIZE and REFRESHWINDOW from the IDCMP Flags variable. Then make the additions shown in Listing 4.10 and compile the program.

Now when you run the program, the window will be smaller. However, when it is resized, the drawing will appear in its entirety and will require no redrawing.

As mentioned, the default position for the window is to be aligned with the upper-left corner of its superbitmap. You can change that position by adding and using the routine in Listing 4.11.

Because ScrollWindow uses one of the Layers library functions, the Layers library must be open before the function can be accessed. Therefore you also have to add, near the beginning of the program, the code to open the Layers library:

```
/* Layers library base address declaration */

LONG LayersBase;
```

```
/* superbitmap declarations (code fragments) */

struct BitMap myBitMap;
extern PLANEPTR AllocRaster();
ULONG *m;

/* added code for superbitmap */

myWindow.Width = 120;
myWindow.Height = 40;          /* shrink it to a superbitmap */

/* allow the superbitmap drawing area to be as large as full-screen */

InitBitMap(&myBitMap,2,640,200);
                  /* depth, width, height */

/* Now allocate memory for the bitplanes associated with the bitmap */

m = AllocRaster(640,400);

if(m == 0)
{
        printf("No memory for superbitmap\n);
        exit(30);
}
myBitMap.Planes[0] = m;          /* that takes care of the first of two */

m = AllocRaster(640,400);

if(m == 0)
{
        printf("No memory for superbitmap\n);
        FreeRaster(myBitMap.Planes[0]);
        exit(30);
}

myBitMap.Planes[1] = m;          /* and this finishes memory allocation */

/* Now that the superbitmap is ready to use, you can attach it */
/* to the NewWindow structure */

myWindow.BitMap = &myBitMap;
```

Listing 4.10: Superbitmap window routines

```
/* opening the Layers library, install this code just */
/* after the opening of the Intuition library */

LayersBase = OpenLibrary("layers.library",0);

if(LayersBase == 0)
{
   printf("Layers library won't open!\n");
   CloseLibrary(IntuitionBase);
   CloseLibrary(GfxBase);
   exit(40);
}
```

```
/* scrollwindow.c */

ScrollWindow(wi,dx,dy);
        struct Window *wi;
        SHORT dx, dy;
{
        struct RastPort *ra;
        struct LayerInfo *li;
        struct Layer *l;

        if(ra = wi->RPort)              /* set RastPort pointer */
        {
            if(l = ra->Layer)           /* set Layer pointer */
            {
                if(li = l->LayerInfo)   /* set LayerInfo pointer */
                {
                    ScrollLayer(li,l,dx,dy);
                }
            }
        }
}               /* end of ScrollWindow */
```

Listing 4.11: The scrollwindow routine

Then, at the end of the program, where the other libraries are closed, install the code

```
CloseLibrary(LayersBase);
```

ScrollLayer is used to reposition an existing layer against its assigned superbitmap layer. If you specify a scroll value that exceeds the maximum scroll distance, the system automatically limits the movement to the maximum that is available. Try adding the following code to your program to move your window against the larger drawing area that the superbitmap uses:

```
for(i = 0; i<40; i + +)
{
   ScrollWindow(w,i,i);                    /* move it diagonally */
   Delay(5);                          /* delay 1/10th of a second */
}
for(i = 39; i>0; i + +)
{
   ScrollWindow(w,i,i);
   Delay(5);                                /* move it back again */
}
```

The Layers library is used by Intuition extensively to create, move, resize, and depth-arrange the windows you create. This book does not go into the other functions of the Layers library. For more information

about that library, see Chapter 5 of Eugene P. Mortimore's *Amiga Programmer's Handbook*, vol. 1 (SYBEX, 1987).

In Chapter 5 of this book, you'll find proportional gadgets described. You may choose to use such a gadget to control what part of the superbitmap you see within your window. But for now, on with graphics.

DESIGNING AND OPENING A CUSTOM SCREEN

In this section, you'll create a custom screen, that is, a screen wherein the choice of colors is yours. Instead of being restricted to the four color choices that the Workbench provides, you can specify your own set of up to 32 colors. On your custom screen, you'll create a map utilizing a number of Amiga graphics-rendering functions.

You must specify the following parameters to enable the system to open a custom screen for you:

- Where to put the screen relative to the overall viewable on-screen area (where to put its upper-left corner)
- How large to make it (how wide, how tall)
- What default title it should have in its title bar
- What colors to use to draw the border and system gadgets
- The characteristics of the font that the menus, window titles, and so forth should use
- The type of screen (in this case, CUSTOMSCREEN) that you want to create
- How many colors this screen should be capable of displaying
- What particular type of viewing mode is to be used

Defining a Custom Screen

The parameters shown above must be defined in a data structure called NewScreen. Listing 4.12 is the NewScreen structure definition that will be used in the map program.

```
/* myscreen1.h */

/* myfont1 specifies characteristics of the default font; */
/* an 80-column font that displays as */
/* 40 columns in low-resolution mode. */

struct TextAttr myfont1 =
{
        "topaz.font", 8, 0, 0
};

struct NewScreen myscreen1 =
{
        0, 0,               /* LeftEdge, TopEdge ... where to put screen */

        320, 200,           /* width, height ... size of the screen */

        5,                  /* 5 planes depth, means 2^5 or */
                            /* 32 different colors to choose from once */
                            /* the screen is opened. */

        1, 0,               /* DetailPen, BlockPen */

        0,                  /* ViewModes ... value of 0 = low resolution */

        CUSTOMSCREEN,       /* type of screen */

        &myfont1,           /* Default font for this screen */

        "32 Color Test",    /* DefaultTitle for its title bar */

        NULL,               /* screen's user-gadgets, always NULL, ignored */

        NULL                /* address of custom bitmap for screen, */
                            /* not used in this example */

};
```

Listing 4.12: The myscreen1 structure definition

Opening the Custom Screen

You open your custom screen by using the OpenScreen function. Here is a typical call to that function:

```
struct Screen *s;              /* declare a pointer to a Screen structure */

s = OpenScreen(&myscreen1);                                  /* try to open it */

if(s == 0)
{
   printf("Can't open myscreen1\n");
   exit(10);
}
```

If this function returns a value of 0, the screen did not open. Assuming that you've correctly specified the various parameters in the NewScreen data

structure, the most common reason that a screen cannot open is insufficient system memory. You may be trying to run too many applications programs at the same time and may need to close down one or more to provide enough memory for this program to run.

Opening a Window on the Custom Screen

Listing 4.13 is a modified version of the NewWindow structure defined earlier in the chapter.

The code below redefines a few of the variables, thereby also making it obvious what is being changed and why. Within the code, once a screen has been opened, the screen pointer in the NewWindow structure must be changed as well, so as to tell the system in which screen it is to open.

```
myWindow.Screen = s;                    /* point to the custom screen */
                                /* and tell system it isn't WBENCHSCREEN */
myWindow.Type = CUSTOMSCREEN;

/*Then, use the normal function call and error checking*/
/* for OpenWindow as shown earlier in the chapter */
```

If the window and screen both open successfully, then you'll have both a custom screen and a custom designed window on the display.

Selecting Colors

Now that you have your own custom screen, you will want to specify the colors that you'll be able to choose from when drawing. Instead of being limited to the colors that Workbench uses, you now have your own custom color palette to modify as you wish for this application.

The NewScreen data structure specified that there should be 32 color choices. You establish your own color palette by using the SetRGB4 function for single colors and the LoadRGB4 function for multiple colors to preset in a single function call.

To use either of these functions, you must retrieve, from the Screen data structure, a pointer to the viewport that the system initialized for your screen when the screen was opened. Here is the code that establishes a pointer to the screen's viewport:

```
struct ViewPort *vp;

vp = &(s->ViewPort);                    /* screen contains a viewport */
```

Now this pointer to a viewport can be used for either SetRGB4 or LoadRGB4. A typical call to SetRGB4 takes the form

```
SetRGB4(vp,colorNumber,rValue,gValue,bValue);
```

```
/* window2.h */

struct NewWindow myWindow =
{
        0,                  /* LeftEdge for window measured in pixels, */
                            /* at the current horizontal resolution, */
                            /* from the leftmost edge of the screen. */
        15,                 /* TopEdge for window measured in lines */
                            /* from the top of the current screen. */
        280, 150,           /* width, height of this window */
        0,                  /* DetailPen - what pen number is to be */
                            /* used to draw the borders of the window. */
        1,                  /* BlockPen - what pen number is to be */
                            /* used to draw system generated window gadgets */

        CLOSEWINDOW | NEWSIZE | REFRESHWINDOW,        /* IDCMP flags */

        SIMPLE_REFRESH | NORMALFLAGS | GIMMEZEROZERO, /* window flags */

        NULL,
        NULL,

        "Sample Chart",         /* window title */

        NULL,                   /* pointer to screen if not Workbench */

        NULL,                   /* pointer to bitmap if a SUPERBITMAP window */


        10, 10,                 /* minimum width, minimum height */
        320, 200,               /* maximum width, maximum height */

        WBENCHSCREEN            /* type of screen in which to open */

};
```

Listing 4.13: The window2 structure definition

where vp is a pointer to a viewport; colorNumber is the color register to be loaded with the RGB color values described by the remaining parameters; and rValue, gValue, and bValue are the red, green, and blue values (ranging from 0 to 15).

Here are three typical calls, using SetRGB4:

```
/* set background color, i.e. color 0, to black (r = 0, g = 0, b = 0) */
SetRGB4(vp,0,0,0,0);

/* set color number 1 to solid RED */
SetRGB4(vp,1,15,0,0);

/* set color number 2 to white (maximum r,g,b values, all) */
SetRGB4(vp,2,15,15,15);
```

You might use SetRGB4 to cycle the colors within one or more color registers so as to give an animation effect of some kind. LoadRGB4 is

most often used to preset the colors for a custom screen shortly after the screen has been opened.

A call to LoadRGB4 takes the form

```
LoadRGB4(vp,colorTable,howmany);
```

where vp is a pointer to the viewport; colorTable is a pointer to the table of colors to be loaded; and howmany tells how many colors there are in the table.

LoadRGB4 always starts its loading with color number zero and proceeds to load as many registers as you specify. The colors are each formulated as unsigned 16-bit numbers with four bits reserved for each of the colors. The bits are arranged as

```
0000 RRRR GGGG BBBB
```

where the first four bits are ignored, the next four bits represent the red value, the next four bits the green value, and the last four bits the blue value.

You can load up to 32 color values into this new custom screen's viewport. Here is an example colortable, containing a total of 16 values. The names of the colors that each represents is shown in the comments.

```
UWORD mycolortable[] = {

0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df
};

/* black, red, white, fire-engine red, orange, yellow, */
/* lime green, green, aqua, dark blue, purple, */
/* violet, tan, brown, gray, skyblue */
```

The function call to use this new color palette for your custom screen is

```
LoadRGB4(vp,&mycolortable[0],16);
```

It will be used in the complete map program listing.

Determining the Colors Currently in Use

The system maintains a colortable for each screen it creates. If you do not load your own colortable, or if you do not load the full 32 values, the system will automatically load them for you from a default color palette when your screen is opened. If the screen was opened by Intuition (i.e., by a call to OpenScreen), you can go into the screen's viewport to determine the color

assigned to a particular color register in the colortable. The routine provided is GetRGB4. A call to GetRGB4 takes the form

```
value = GetRGB4(vp,entry);
```

where vp is a pointer to the address of the screen's viewport, and entry is the color register number in which you are interested.

The value returned is −1 if that entry number does not contain a valid value or the actual RGB value with four bits assigned for each red, green, and blue intensity, where the most significant four bits of the returned value contain 0, the next four bits contain the red value, the next four bits contain the green value, and the least significant four bits contain the blue value.

Flood-Filling Shapes

The Amiga includes the Flood routine to fill a shape with a color beginning at a specified position in the drawing area. There are two modes to this flood-filling. Mode 0 fills a shape with color starting at the current pen location and spreading out to all adjacent pixels, not stopping until a color the same as the area-outline pen (the color set by the call to the SetOPen function) is reached. Mode 1 fills a shape with color starting at the current pen location and filling all adjacent pixels that are the same color as the one on which the flood-fill began.

The interesting thing about the Flood function is that it uses the current drawing modes (JAM1 or JAM2) and the pattern, if any. So you can form an enclosed area of any shape, and flood-fill it with a multicolored pattern if you wish. If you use this function, you'll need to be sure that the shape you wish to fill has no breaks in its outline. Any time there is at least one horizontally or vertically adjacent pixel available, the fill can leak out and fill your entire window or screen area with the fill pattern.

A call to Flood takes the form

```
Flood(rp,mode,x,y);
```

where rp is a pointer to a rastport; mode is the mode number; and x and y are the coordinates at which to start the flood-fill.

Listing 4.14 uses the Flood function in a routine to create a filled diamond shape.

Oddly Shaped Filled Areas

In addition to the Flood function, the Amiga software provides a series of functions for area-filling. With these functions, you define the entire shape first, then tell the system to fill it, rather than drawing a series of lines and asking for a flood-fill. The advantage to the area-filling

```
/* drawdiamond.c */

DrawDiamond(rport,xcenter,ycenter,xsize,ysize)
        struct RastPort *rport;
        WORD xcenter, ycenter, xsize, ysize;
{
        BYTE oldAPen;
        WORD xoff, yoff;

        oldAPen = rport->FgPen;         /* save old value of APen */

        SetAPen(rport,rport->AOlPen);
                                        /* same color as outline pen */

        xoff = xsize/2;                 /* offsets from the center */
        yoff = ysize/2;

        /* draw the diamond shape */

        Move(rport,xcenter - xoff,ycenter);
        Draw(rport,xcenter,ycenter + yoff);
        Draw(rport,xcenter + xoff,ycenter);
        Draw(rport,xcenter,ycenter - yoff);
        Draw(rport,xcenter - xoff,ycenter);

        /* flood-fill it from its center */

        Flood(rport,0,xcenter,ycenter);     /* out to outline pen color */

        SetAPen(rport,oldAPen);             /* restore value of APen */

}
```

Listing 4.14: The drawdiamond routine

functions is that they automatically handle any strange shape that you might define, with no danger of a gap being left causing the fill to leak out of the intended drawing area.

Here are the functions that are used for area-filling:

```
error = AreaMove(rp,x,y);
error = AreaDraw(rp,x,y);
AreaEnd(rp);
```

The rp parameter is a pointer to a rastport, and x and y are the coordinates for a move or draw.

A value of −1 is returned if there was no room left in the buffer to hold this particular AreaMove or AreaDraw request; if there was no error, 0 is returned.

AreaMove operates similarly to Move in that it effectively means "pick up the drawing pen and move it somewhere else." AreaDraw operates similarly to Draw in that it means "draw an outline (with or without an area-outline pen for the final shape) from the current pen position to the new coordinate location." Neither AreaMove nor AreaDraw has any

effect on the current position of the drawing pen used for Move and Draw.

You can have several AreaMove and AreaDraw functions, each defining a separate shape. When you finally call AreaEnd, all of the shapes that you defined are drawn at once. No shapes or lines are drawn until AreaEnd is called.

When you call AreaMove, any prior shape being created by a series of AreaDraw function calls is automatically closed as though you had called AreaDraw one more time, specifying the coordinates of the very first point in that shape. For example, to draw a filled square, you need only call AreaMove to move to the first corner and call AreaDraw to draw to the other three corners. Your next call to AreaMove (or AreaEnd), automatically draws the closing side to the square.

Prerequisites for Area Operations

As you call the AreaMove and AreaDraw functions, the system builds a list of the move and draw operations. When you call AreaEnd, the system processes this list, drawing and filling the shapes. There are a couple of things you must do prior to calling these functions to prepare your rastport for the system to use. You must provide the following:

- An AreaInfo data structure for this rastport and a corresponding data area for the AreaInfo structure
- A TmpRas data structure for this rastport and a corresponding memory workspace

The AreaInfo data structure contains variables that the system uses to keep track of your requests for AreaMove and AreaDraw functions. It also contains a pointer to a memory space that is used to save the actual requests. Recall that no area-filling actually takes place until you call AreaEnd. All of your intermediate requests to AreaMove and AreaDraw get stored in a memory array that you provide, and all get executed at one time when AreaEnd is called.

You must provide an array of 16-bit words that the system can use for storing your request. The array must contain five times as many words as the total number of calls to AreaMove and AreaDraw that you wish to make before calling AreaEnd. If you are creating your shapes one at a time, you need only provide as many points as there are vertices in the shape you are creating (plus a couple more for good measure, perhaps). For this example, lets provide for a maximum of 20 points. You initialize an AreaInfo structure by using the InitArea function, passing it the address of your AreaInfo structure, the address of

the array into which the system will store your requests, and the maximum number of points to allow:

```
WORD areaArray[100];          /* 20 points times 5 words per point */
struct AreaInfo myAreaInfo;
InitArea(&myAreaInfo,&areaArray[0],20);
rp->AreaInfo = &myAreaInfo;          /* link it to the rastport */
```

When the system begins to fulfill your request for an area-fill, it requires a work area in which to build and fill the shape before the shape is moved into the actual drawing area that you have specified. The shape is built in what is called the TmpRas, or temporary raster area.

The number of data words of space needed for a TmpRas data structure is defined by the largest rectangular shape you wish to produce. For example, if the largest shape is 200 lines tall by 640 lines wide, then you need a space large enough to accommodate 640-by-200 bits. Only the physical size of the multiplied width and height is taken into account. That is, the memory is reused, for each area-fill operation, in a manner appropriate to the shape of the object and not constrained to the rectangular coordinates on which the memory allocation was based. For example, if you allow a space large enough to draw a shape that fits within a 100-by-100 rectangle, then it can just as easily handle a shape that fits in a 1000-by-10 rectangle.

You must allocate the workspace memory first, then use the InitTmpRas function to initialize the TmpRas data structure, then link it into your RastPort. Here is a typical sequence:

```
PLANEPTR workspace;
struct TmpRas myTmpRas;

workspace = AllocRaster(640,200);
if(workspace = = 0)
{
   printf("No space for Temporary Raster!\n");
}
else
{
   InitTmpRas(&myTmpRas,workspace,RASSIZE(640,200));
   rp->TmpRas = &myTmpRas;          /* link it into the rastport */
}
```

The macro function, RASSIZE, is used to tell the system how many data words have been reserved by the AllocRaster call. The system then

knows how large a shape can be produced using this structure.

Before using the Text function, you should provide a TmpRas structure for your rastport, because if Text does not have this structure, it must allocate and deallocate workspace memory every time it is called. Providing a TmpRas is more efficient and therefore can make the system run faster. Using the Text function is discussed later in this chapter.

Drawing and Reading Individual Pixels

Sometimes, you will want to either draw into or find out the color of an individual pixel (picture element). The Amiga graphics routines include WritePixel and ReadPixel for these purposes. The calls to these routines take the form

```
WritePixel(rp,x,y);
penNumber = ReadPixel(rp,x,y);
```

where rp is a pointer to a rastport; and x and y are the coordinates at which to read or write.

For WritePixel, the color that is drawn is that of the APen. For ReadPixel, the value returned, penNumber, ranges from 0 to 255. If it is not possible to read the selected coordinates (perhaps they are outside the range of the X or Y coordinates for this rastport), a value of −1 is returned.

In the map program that follows, WritePixel is used with the Amiga's random number function to add some action to the map.

Drawing a Map

The map program in Listing 4.15 produces a map of Utah, Colorado, Arizona, and New Mexico. It shades each state with a separate color, then labels each state with a two-letter abbreviation of its name. Once the states are drawn, the program "snows" on Arizona. To start the snowing effect, click the left mouse button in the window, then click the right mouse button.

TEXT

Thus far, you've used the default 80-column font for all of the displayed text. Now you'll learn how to create special effects for the text and how to use the other available fonts.

To move the cursor correctly regardless of the type of font present, you need to know how to find out the height, width, and baseline of the text. And to use different fonts, you need to know how to switch fonts

```
#include "exec/types.h"
#include "intuition/intuition.h"

#define AZCOLOR 1        /* The color numbers to use as Arizona and */
#define WHITECOLOR 2     /* as snow */

#include "mydefines.h"   /* gets the window flags assignments */

#include "myscreen1.h"
#include "window2.h"
#include "graphics/gfxmacros.h"
#include "event1.c"      /* gets the event handler */

/* define the initial placements of the 4 corners of the states */

#define CORNERX 150
#define CORNERY 75

struct Window *w;        /* pointer to a window */
struct RastPort *rp;     /* pointer to a rastport */
struct Screen *s;        /* pointer to a screen */
struct ViewPort *vp;     /* pointer to a viewport */

struct AreaInfo myAreaInfo;
PLANEPTR workspace;
struct TmpRas myTmpRas;

struct Window *OpenWindow();
struct Screen *OpenScreen();

LONG GfxBase;
LONG IntuitionBase;

WORD areaArray[100];        /* 20 points times 5 words per point */

WORD utahxy[] =
        {
        0, 0, -40,0,  -38,-70,  -15,-70,  -17,-55,  0,-55, 0,0
        };

WORD coloradoxy[] =
        {
        0,0,   75,0,  75,-55,  0, -55
        };

WORD arizonaxy[] =
        {
        0,0,  -40,0,  -40,10, -50,10,  -50,30,
        -60,55,  -30,70,  0,70
        };

WORD newmexicoxy[] =
        {
        0,0,   0,70,   8,70,   68,70,   68,0
        };

UWORD mycolortable[] =
        {
        0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
        0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
        0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df
        };

/* black, red, white, fire-engine red, orange, yellow, */
```

Listing 4.15: The map program

```
/* lime green, green, aqua, dark blue, purple, */
/* violet, tan, brown, gray, skyblue */

#include "drawdiamond.c"

main()
{
        struct IntuiMessage *msg;
        LONG result;
        PLANEPTR workspace;
        WORD rx, ry;

        GfxBase = OpenLibrary("graphics.library",0);
        IntuitionBase = OpenLibrary("intuition.library",0);
        /* (error checking left out for brevity here) */
        /* (should check that neither OpenLibrary returned 0) */

        s = OpenScreen(&myscreen1);  /* try to open it */
        if(s == 0)
        {
                printf("Can't open myscreen1\n");
                exit(10);
        }
        myWindow.Screen = s;        /* say where screen is located */
        myWindow.Type = CUSTOMSCREEN;
                        /* tell Intuition to look at w.Screen */
        myWindow.Title = "Sample Map";

        w = OpenWindow(&myWindow);
        if(w == 0)
        {
                printf("Window didn't open!\n");
                CloseScreen(s);
                exit(20);
        }
        vp = &(s->ViewPort);

        /* set the colors for this viewport */

        LoadRGB4(vp,&mycolortable[0],16);

        rp = w->RPort;
        workspace = (PLANEPTR)AllocRaster(640,200);
        if(workspace == 0)
        {
                printf("No space for Temporary Raster!\n");
                CloseWindow(w);
                CloseScreen(s);
                exit(30);
        }
        InitTmpRas(&myTmpRas,workspace,RASSIZE(640,200));
        rp->TmpRas = &myTmpRas;        /* link it into the rastport */

        InitArea(&myAreaInfo,&areaArray[0],20);
        rp->AreaInfo = &myAreaInfo;        /* link it to the rastport */

        redraw();        /* for the "first" time */

        /* Now wait for a message to arrive from Intuition */
        /* (task goes to sleep while waiting for the message) */

        WaitPort(w->UserPort);
```

Listing 4.15: The map program (continued)

```
        /* retrieve the message from the port */

        while(1)        /* "forever" */
        {
                msg = (struct IntuiMessage *)GetMsg(w->UserPort);
        handleit:
                result = HandleEvent(msg->Class);

                if(result == 0)             /* got a CLOSEWINDOW */
                    break;
                msg = (struct IntuiMessage *)GetMsg(w->UserPort);


                if(msg != 0)                /* will be 0 when */
                                            /* no more messages */
                    goto handleit;
                /* Normally, to correctly interface with the multi- */
                /* tasking, there would be a statement here to the */
                /* effect                                          */
                /*                                                 */
                /*          WaitPort(w->UserPort);                 */
                /*                                                 */
                /* to put the task to sleep awaiting another */
                /* message since by now we've handled all of */
                /* them. But in this case, the task wants to */
                /* still be busy snowing-on-Arizona so the */
                /* WaitPort has been left out. */

                /* Arizona exists as an odd shape within a rectangle */
                /* based at (0,0), going to (-60,70). We want to */
                /* pepper the state itself with snow without dropping */
                /* any snow outside of the outline of the state. */
                /* Here, let's pick a random x and y position, then */
                /* use ReadPixel to see if it falls on the color */
                /* that was used to fill the state. If in range, */
                /* use WritePixel to write a white dot. */

                rx = CORNERX - RangeRand(60);
                ry = CORNERY + RangeRand(70);

                if(ReadPixel(rp,rx,ry) == AZCOLOR)
                {
                        SetAPen(rp,WHITECOLOR);

                        WritePixel(rp,rx,ry);
                }
        }
        /* done! Now cleanup. */

        CloseWindow(w);
        CloseScreen(s);
        FreeRaster(workspace,640,200);
}

afill(w, pairs)
        WORD *w;        /* pointer to a word */
        WORD pairs;     /* how many pairs of words */
{
        WORD i;
        AreaMove(rp,CORNERX+w[0],CORNERY+w[1]);
        w++; w++;
        for(i=1; i<pairs; i++)
        {
```

Listing 4.15: The map program (continued)

```
                AreaDraw(rp,CORNERX+w[0],CORNERY+w[1]);
                w++; w++;
        }
        AreaEnd(rp);
}

redraw()
{
        SetDrMd(rp,JAM1);
        SetAPen(rp,1);
        afill(&coloradoxy[0],4);

        SetAPen(rp,5);
        afill(&utahxy[0],7);

        SetAPen(rp,3);
        afill(&newmexicoxy[0],5);

        SetAPen(rp,AZCOLOR);
        afill(&arizonaxy[0],8);

        SetOPen(rp,12);
        SetAPen(rp,7);
        DrawDiamond(rp,20,20,20,10);
        SetAPen(rp,8);
        DrawDiamond(rp,20,120,20,10);

        SetAPen(rp,6);
        SetBPen(rp,0);
        SetDrMd(rp,JAM2);


        /* label the states */
        Move(rp,CORNERX-30,CORNERY-20);
        Text(rp,"UT",2);

        Move(rp,CORNERX-30,CORNERY+30);
        Text(rp,"AZ",2);

        Move(rp,CORNERX+35,CORNERY-20);
        Text(rp,"CO",2);

        Move(rp,CORNERX+35,CORNERY+30);
        Text(rp,"NM",2);
}
```

Listing 4.15: The map program (continued)

and how to ask for normal, bold, italics, or inverse video. These are among the topics covered in this section.

The height of the text can be found in the RastPort structure variable named TxHeight. If you have a pointer to a rastport named rp, then you access the text height by

```
myTextHeight = rp->TxHeight;
```

The nominal width of the text, that is, that width of rectangle into which all regular text (not bold, not italics) will fit is accessed by

```
myTextWidth = rp->TxWidth;
```

You may wish to use the values of the text height and width to produce a line or box cursor of the same size as the text on which you are positioning the cursor.

The baseline of the text is stored in the RastPort structure variable TxBaseline. You access this value by

```
myTextBaseline = rp->TxBaseline;
```

The baseline is the imaginary line that positions the text. Some fonts have descenders, so that characters such as y and g extend below the bottommost line of other characters such as A. The baseline position is measured from the topmost line of the standard character cell (the text width and height).

To position a character with the upper-left corner of its character cell at X,Y, you must perform the function

```
Move(rp,X,Y + baseline);
```

Then, when you use the Text function, the character will be drawn as you expect.

Opening a Font

There are two different kinds of fonts: ROM resident and disk resident. The Amiga ROM software provides two different routines for accessing these fonts. One is OpenFont, for the ROM-resident fonts. The other is OpenDiskFont, for those on disk. In practice, however, the OpenDiskFont routine is the only routine that you need to use, because it can handle both ROM fonts and disk fonts. It simply uses the information contained in the system font list to determine where to find a font and whether its data is already loaded or needs to be loaded.

If you wish to use a font other than the 60-column or 80-column font you've selected by Preferences, or a font other than the default font that you specified in your NewScreen structure, you must open the font and you must select that opened font for your rastport.

Opening a ROM-resident font simply makes it available for use, returning to your calling routine a pointer to the data structure that describes and controls the font. Opening a disk font performs that same action, but in addition, loads the font from disk if it is present in the current FONTS directory. Thus, once a font has been opened, it becomes available for use.

The calls to OpenFont and OpenDiskFont take the form

```
font = OpenFont(&ta);
font = OpenDiskFont(&ta);
```

where font is a pointer to a TextFont data structure returned by the routines if it was possible to open this font; and &ta is a pointer to a TextAttr data structure.

Defining Text Attributes

The name, size, and style of each font are contained in the TextAttr structure variables:

- ta_Name is the address of a null-terminated string that describes the name of the font as it appears in the FONTS directory (for example, garnet).
- ta_YSize is the nominal height of the font in lines. In the font name subdirectory of the FONTS directory there are file names that match the YSize entries for the font (for example, garnet/9 is a 9-line-high version of the garnet font). This entry contains the font itself in binary format.
- ta_Style is the primary style of the font. Some fonts will be italic or bold and are stored as such. Other fonts can be made to look bold or italic.
- ta_Flags includes the font Preferences. Font Preferences are those flags that you can set to ask the system to match particular requests as well as it can. For example, if you wanted to use a font that was designed for high-resolution, noninterlaced displays, you might set the flag FPF_TALLDOT. If there are two fonts in the system having the same name and same YSize, but differing in that one was designed for high-resolution, it is the high-resolution font that will be loaded.

Here is an example of a text-attribute structure:

```
struct TextAttr myAttr = { "garnet.font",9,0,0 };
```

Here is an example of a call to OpenDiskFont:

```
struct TextFont *tf;

tf = OpenDiskFont(&myAttr);
```

Establishing the Rastport's Font

You specify which font the rastport should use for Text calls by using the SetFont routine. A call to SetFont takes the form

```
SetFont(rp,tf);
```

where rp is a pointer to a rastport, and tf is a pointer to a TextFont data structure returned from OpenFont or OpenDiskFont. After setting the font, any calls to Text will use this font.

Adding a Font to the System Font List

There are two fonts already on the system font list (selectable from Preferences), namely topaz.font 8-lines high (the 40/80 column font) and topaz.font 9-lines high (the 32/64 column font). For other fonts, if you want them to be accessible to all tasks running concurrently, you should add them to the system font list by using AddFont. A call to AddFont takes the form

```
AddFont(tf);
```

where tf is a pointer to a TextFont data structure returned from a call to OpenFont or OpenDiskFont.

If two tasks need to use the same font, each will call OpenDiskFont to get the TextFont pointer. The first thing the system will do in this case is to look at the system font list to see if another task has already loaded this font. If so, then the system adds one to the number of current users of the font and returns the pointer to the TextFont structure to the caller. If the font has not been added to the system font list, then all of the data of the font will be loaded again, taking up unnecessary space.

Which Fonts Are Available

Normally, a programmer will know which fonts are available to be used and can ask for specific ones by name and by physical characteristics. The OpenFont and OpenDiskFont routines attempt to match a majority of the font Preferences and may return something close to the expected font if that font is not available in the FONTS directory. For example, if you specify garnet.font in a 9-high size, and if it is not present, you may get some other font in 9 high. You should check the font's characteristics (using AskFont) after calling OpenFont or OpenDiskFont to see exactly what the system has found for you.

You can create a list of fonts that meet certain type characteristics by using a routine called AvailFonts. Using this routine, you can, for example, list all of the fonts that are ROM-based, or fonts that are proportionally spaced, or other possible combinations of types.

A call to AvailFonts takes the form

```
AvailFonts(&afh,AFSIZE,types);
```

where &afh is a pointer to an area in memory to hold the AvailFonts-Header structure and a series of AvailFonts structures; AFSIZE tells the

system how large an array you've allocated to hold both the header and all of the entries; and types is a single byte containing bits that indicate which types of fonts you wish included in this list. If there are more fonts available than space to list them in, the system will stop loading fonts before overflowing the space you've allocated.

A types value of 0xff lists all the fonts. You can find a full list of the font flags in graphics/text.h. Here are some flags that you might set:

- FPF_ROMFONT—lists those fonts that are located in ROM.
- FPF_DISKFONT—lists those fonts that are located on disk.
- FPF_REVPATH—lists the fonts that are designed to be rendered from right to left instead of left to right (Hebrew, for example).
- FPF_WIDEDOT—lists the fonts designed for low-resolution, interlaced displays.
- FPF_TALLDOT—lists the fonts designed for high-resolution, non-interlaced displays.

You select combinations of characteristics by performing an OR operation on individual flags. Once you have generated this list, you'll be able to look through it for fonts that have only the characteristics you want.

AvailFonts automatically calls AddFont for each of the fonts you've listed that are not already on the system font list. This has an interesting side effect in that if you call AvailFonts a second time, requesting all types, you'll find that the system will list the disk-resident fonts twice in your AvailFonts data structure. Because AvailFonts calls AddFont the first time you run it, the font appears on the system list as well as in the FONTS directory. Thus it will get listed twice.

Knowing that there are currently only two fonts that are ROM-resident, you might consider not setting the FPF_ROMFONT flag, and you'll wind up with a list of fonts that are available exclusively from disk when the routine returns.

After AvailFonts returns, the system will have filled in your AvailFonts header and AvailFonts arrays for you so you'll know which fonts you can use and what are their characteristics. Here is how the AvailFonts-Header structure and its associated AvailFonts data structures are laid out in memory:

```
AvailFontsHeader    { UWORD afh_NumEntries }
AvailFonts          { UWORD af_Type;
                              /* 1 = mem.res, 2 = disk */
                    struct TextAttr af_Attr;
```

```
                                    }                          /* attributes */
        AvailFonts
        <more>
```

AvailFontsHeader simply contains the number of AvailFonts entries that follow it. AvailFonts consists of the entry type (memory or disk) followed by a TextAttr data structure.

Once you have called AvailFonts, you can establish a pointer to any one of the AvailFonts entries in the table and pass this pointer value to OpenDiskFont, as illustrated below.

To allocate enough memory for your arrays for the AvailFonts call, you might use the following sequence:

```
#define AF_NUMENTRIES 30

struct AvailFontsHeader *aftable;
struct aft =
{
                    struct AvailFontsHeader aft_Head;
                    struct AvailFonts aft_Entry[ AFNUMENTRIES ];
};

int aftablesize;

aftablesize = sizeof(struct aft);
```

Then call AvailFonts by

```
AvailFonts(&aft,aftablesize,0xFE);
```

where the types variable is set to 0xFE to exclude listing the fonts that are ROM based. The text attributes for the ROM fonts are as follows and can be used separately if you wish:

```
struct TextAttr size32_64 = { "topaz.font",9,0,0 };
struct TextAttr size40_80 = { "topaz.font",8,0,0 };
```

Listing 4.16 provides code fragments for extracting and using information from the available fonts table.

Text Characteristics

You can control how fonts are rendered into your rastport. The attributes you can control are color, (including inverse video), boldfacing, italicizing, and underlining.

You select text color by using the SetAPen, SetBPen, and SetDrMd functions. The body of the text is drawn with the APen color. If the

```
/* extract information from the fonts table */
UWORD howmany_entries;

struct TextAttr *myTextAttrPointer;

howmany_entries = aft.aft_Head.afh_NumEntries;

/* point to the first entry in the array of AvailFonts */

myTextAttrPointer = (struct TextAttr)(&aft.aft_Entry);

/* use these entries */

int i;
struct TextFont *OpenDiskFont(), *tf;

for(i = 0; i < howmany_entries; i++)
{
        tf = OpenDiskFont(myTextAttrPointer[i]);

        if(tf != NULL)          /* if the font opened.... */
        {
                SetFont(rp,tf);

                /* rp is a pointer to a rastport that */
                /* you want to use for this font when any */
                /* text is being output */

                Text(rp,"sample text",11);
                Delay(30);
                CloseFont(tf);          /* close what you open */
        }
}
```

Listing 4.16: Font-related code fragments

drawing mode is set to JAM1, this is the only color that is rendered when a text character is drawn. The enclosing rectangle of the text is drawn with the BPen color if the drawing mode is set to JAM2.

Text highlighting is often achieved by inverting the colors that are used to render the text. In other words, the APen is used to render the background, and the BPen is used to render the text. You need not swap the pen values to accomplish this. You need only set the drawing mode (SetDrMd) to include INVERSVID as shown here:

```
SetDrMd(rp,JAM1 + INVERSVID);          /* render the text in the */
                                    /* background color against a */
                                       /* rectangle that is entirely */
                                          /* the primary pen color */

SetDrMd(rp,JAM2 + INVERSVID);          /* render the text in the */
                                    /* background color and the */
```

```
/* background in the primary */
/* pen color */
```

Bold, Italic, and Underlined Text

You select these characteristics by using the SetSoftStyle function. You can ask the system to adjust a current font to render its characters in one or more of these particular styles. A call to SetSoftStyle takes the form

```
SetSoftStyle(rp,style,mask);
```

where rp is a pointer to a RastPort; style is a value containing bits representing the style you wish to set; and mask is a set of bits that tells the system exactly which of the style bits you want to affect.

The mask value is useful if, for example, you have selected italics and now want to turn on underline mode without affecting the italics setting. (The flag bits are specified in the system Include file graphics/text.h.) You can set underlining without affecting any other bits by specifying the same value for both the mask and the style value, such as

```
SetSoftStyle(rp,FPF_ITALICS,FPF_ITALICS);
```

If you want to reset all of the style bits regardless of what they are currently set to, use a value of 0xFF for the mask:

```
SetSoftStyle(rp,FPF_ITALICS,0xFF);

/* don't care if it was bold and underlined and so on... */
/* just make it italicized. */
```

If a font has already been designed as boldfaced, then you ask the system to render it boldfaced, no further bolding will happen since the bold bit will already be a part of the basic style of the text itself (in the font flags where the font is described). The same holds true for italics and underlining. If you wish to determine which characteristics you can legitimately set for a current font, use the AskSoftStyle function. A call to AskSoftStyle takes the form

```
enable = AskSoftStyle(rp);
```

where rp is a pointer to a rastport. The value returned, enable, is a data byte containing the flags that show you which algorithmically generated styles you can choose for this font.

Here's a program fragment that lists those styles:

```
UBYTE enable;

enable = AskSoftStyle(rp);
```

```
if(enable & FPF_UNDERLINED)
        printf("OK to ask for underlining\n");
if(enable & FPF_BOLD)
        printf("OK to ask for bold\n");
if(enable & FPF_ITALICS)
        printf("OK to ask for italics\n");
```

Here's a final note about underlining. When the system generates an underline for text, it places it on the line below the baseline of the text. Some fonts may be designed with no part of any character extending below the baseline of the text. The system cannot render an underline for this kind of a font since the underline would have to be rendered outside of the enclosing rectangle of the text itself. You can tell if underlining is possible by comparing the text height with the text baseline value for the text in your rastport. Here is one way to determine that:

```
if( rp->TxHeight - rp->TxBaseline == 1)
        printf("Won't be able to underline this one!");
```

The top line of text is line zero, thus the numerical value of the bottommost line is TxHeight minus one.

Clearing and Scrolling Drawing Areas

You can clear an entire rastport drawing area by using the SetRast function. This function sets an entire rastport's drawing area to a single color. A call to SetRast takes the form

```
SetRast(rp,color);
```

where rp is a pointer to a rastport, and color is the color to which it is to be set.

You can use this function on the rastport you obtain from Intuition when you open a window. However, if you don't specify GIMMEZEROZERO for the window flags, using SetRast will erase all of your window's borders and gadgets as well as the internal drawing area. If you use GIMMEZEROZERO, only the drawing area will be affected.

The ClearEOL and ClearScreen functions are available in the Graphics library. These are text-oriented functions and are used by the Console device to clear to the end of a line and to clear to the end of the screen. Because they are text-oriented, the way they function depends on the text font that is currently selected for the rastport.

ClearEOL starts at the current drawing pen position and clears a rectangular area to the rightmost edge of the rastport as tall as the RastPort structure TxHeight variable setting and positioned just as text

would be if text were being drawn. ClearScreen performs ClearEOL, then clears out the area of the rastport below that cleared line.

The calls to these functions take the form

```
ClearEOL(rp);
ClearScreen(rp);
```

where rp is a pointer to the rastport to be cleared.

When you are using text for things such as terminal programs or word processors, you may wish to scroll up a series of lines to make room for another line at the bottom or scroll down some lines to make room for another line at the top. A smooth scrolling action is certainly more desirable than redrawing all of the lines on the screen. The ScrollRaster function can be used for this purpose. A call to ScrollRaster takes the form

```
ScrollRaster(rp,dx,dy,xmin,ymin,xmax,ymax);
```

where rp is a pointer to a rastport; xmin, ymin, xmax, and ymax are the coordinates of the upper-left and lower-right corners of a rectangle enclosing the area of the rastport you would want to scroll; and dx and dy are the number of pixels (dx) and lines (dy) that this segregated area should be moved towards the 0,0 coordinates of this area. These may be positive, zero, or negative values.

For example, scrolling up a set of lines to make room for another line at the bottom would use a value of dx equal to 0 (not scrolling either left or right) and dy equal to 8 (8 pixels upwards, towards 0,0).

ScrollRaster comes in handy if you are not using GIMMEZEROZERO windows in that you can limit the scrolling area to that within the boundaries of the window, leaving the borders undisturbed.

COMBINING OBJECTS TO FORM A PICTURE

In this section you will learn about creating your own off-screen drawing area. Up till now, the programs in this chapter have used a rastport obtained from an Intuition call to OpenWindow. Here, you'll see the functions required to create the drawing area and its rastport directly. You will draw a few objects in an off-screen rastport, then copy the entire composite object into a window, where it may then be seen.

To understand how to do this, you need to know

- How to initialize a bitmap and allocate memory for its bitplanes
- How to initialize a rastport

- How to copy data from one bitmap to another

We'll examine each of these in turn.

Initializing a Bitmap

The BitMap data structure contains variables that define the size of a drawing area and the location of the memory that is dedicated for its use. You use the InitBitMap function to initialize the data structure itself, then, for as many planes as the bitmap is to contain, you must allocate some memory that it can use for storing the bits that represent the drawing area.

As an example, say you want to have an off-screen drawing area of 320 by 200, that allows up to four colors. This requires a depth of 2, that is, two bitplanes dedicated to this bitmap. Here is a sample initialization:

```
#define DEPTH 2
#define WIDTH 320
#define HEIGHT 200

struct BitMap myBitM;
int i;
extern PLANEPTR AllocRaster();

InitBitMap(&myBitM,DEPTH,WIDTH,HEIGHT);

for(i = 0; i<DEPTH; i + +)
{
   myBitM.Planes[i] = AllocRaster(WIDTH,HEIGHT);
   if(myBitM.Planes[i] = = 0)
   {
                        /* do error processing.... not enough memory */
   }
}
```

And that's all there is to it. Later, when you are finished using this off-screen bitmap, you must free the memory you've allocated, typically by the following sequence:

```
for(i = 0; i<DEPTH; i + +)
{
   if(myBitM.Planes[i] ! = 0)
   {
        FreeRaster(myBitM.Planes[i],WIDTH,HEIGHT);
   }
}
```

Initializing a Rastport

Now that you have reserved space for the bitplanes, initializing the rastport is easy:

```
struct RastPort myRast;

InitRastPort(&myRast);

myRast.BitMap = &myBitM;        /* link the bitmap into this rastport */
```

Now, using a pointer to this rastport, you can set its drawing pen color numbers and its drawing mode, move the drawing pen, and do everything else you might wish to do with graphics. The next two lines define a pointer to an off-screen rastport and establish the value of the pointer.

```
struct RastPort *offscreenrp;          /* a pointer to this rastport */

offscreenrp = &myRast;
```

In this off-screen rastport, you can draw complex shapes that you might not want the user to see until the shape has been completed. Then you can copy this shape from the off-screen area to an on-screen rastport.

Copying Data from One Bitmap to Another

The Amiga provides three different routines for copying data from one bitmap to another: BltBitMap, ClipBlit, and BltBitMapRastPort.

BltBitMap copies data from one bitmap to another, bypassing the rastport usage completely. This routine has more potential for crashing the system than the other two in that no checking is done to determine whether all of the bits will actually fit into the destination area. Data moves (called blits) outside of the boundaries of the bitmap invariably destroy some data that should not have been touched and thus can crash the system.

ClipBlit copies data from one rastport to another. It cuts out a rectangle of data at a particular X,Y coordinate in a source rastport and places it into a selected X,Y coordinate in the destination rastport. The disadvantage to ClipBlit is that if you have two objects that are intended to overlap each other in the destination area, it is the rectangles that will overlap, not just the objects themselves. If an object is surrounded by some blank space, for example, the blank space of the second object will erase some part of the first object placed into the destination area.

BltBitMapRastPort is slightly faster than ClipBlit because the copy takes place from the source bitmap to a destination rastport. This means there are no layering operations done on the source area and it is

assumed that the source is indeed not composed of overlapping layers.

The call to ClipBlit takes the form

```
ClipBlit(src_rp,src_x,src_y,dest_rp,dest_x,dest_y,
                    size_x,size_y,minterm);
```

The parameters to ClipBlit are as follows:

- src_rp and dest_rp are pointers to the source and destination rastports
- src_x and src_y are the X,Y coordinates of the upper-left corner of the source area enclosing rectangle
- dest_x and dest_y are the X,Y coordinates of the destination area at which to place the rectangle of data to be copied
- size_x and size_y are the horizontal and vertical dimensions of the rectangle to be copied
- minterm is a value that indicates exactly how the data is to be treated as the copy progresses

A minterm value of 0xC0 makes a direct copy from source to destination. A value of 0x30 inverts the source—wherever there was a 0-bit, a 1-bit is placed and vice versa. A value of 0x50 ignores the source data entirely and simply inverts the destination area rectangle.

A call to BltBitMapRastPort takes the form

```
BltBitMapRastPort(src_bm,src_x,src_y,dest_rp,dest_x,dest_y,
                    size_x,size_y,minterm);
```

where all the parameters are the same as those for ClipBlit, except that src_bm is the source bitmap from which the data is to be copied.

Using Data-Move Routines

Listing 4.17 opens one simple-refresh window and an off-screen rastport. In the first window, a few colored lines are drawn. In the off-screen rastport, a set of rectangles is drawn and moved into the on-screen area by ClipBlit and BltBitMapRastPort, each in turn.

In the next chapter, you'll see much more about Intuition, including how to interpret mouse position and mouse button events. For this program, a simple delay controls this operation.

Copying with Transparency

You will have noticed that with both ClipBlit and BltBitMapRastPort the entire rectangle of the copied object is brought along. To avoid this, you

```
/* offscreen.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "mydefines.h"
#include "window1.h"
#include "df1:include/graphics/gfxmacros.h"


#define DEPTH 2
#define WIDTH 640
#define HEIGHT 200

int IntuitionBase, GfxBase;

int i,j;
extern PLANEPTR AllocRaster();

struct BitMap myBitM;                   /* the off-screen area bitmap */
struct RastPort myRast;                 /* the off-screen area rastport */
struct RastPort *offscreenrp;           /* a pointer to this rastport */

struct RastPort *rport;                 /* the on-screen rastport pointer */
struct Window *w;                       /* the on-screen window pointer */
extern struct Window *OpenWindow();

main()
{
        GfxBase = OpenLibrary("graphics.library",0);
        if(GfxBase == 0)
        {
                printf("graphics.library won't open!\n");
                exit(10);
        }
        IntuitionBase = OpenLibrary("intuition.library",0);
        if(IntuitionBase == 0)
        {
                printf("intuition.library won't open!\n");
                exit(15);
        }

        w = OpenWindow(&myWindow);
        if(w == 0)
        {
                printf("Window didn't open!\n");
                CloseLibrary(GfxBase);
                exit(20);
        }
        rport = w->RPort;

        InitBitMap(&myBitM,DEPTH,WIDTH,HEIGHT);

        for(i=0; i<DEPTH; i++)
        {
                    myBitM.Planes[i]=AllocRaster(WIDTH,HEIGHT);
                    if(myBitM.Planes[i] == 0)
                    {
                  /* do error processing.... not enough memory */
                    }
        }
        InitRastPort(&myRast);

        myRast.BitMap = &myBitM;  /* link the bitmap into this rastport */
        offscreenrp = &myRast;
```

Listing 4.17: The offscreen program

```
        /* draw a few lines in the on-screen area so we can */
        /* see what happens when off-screen data gets moved */


        SetAPen(rport,3);
        SetDrMd(rport,JAM1);

        j = 10;
        for(i=0; i<30; i++)
        {
                Move(rport,j,0);
                Draw(rport,j,100);
                j += 10;
        }

        /* draw something into the off-screen area */

        SetRast(offscreenrp,0);         /* blank it out first */

        SetAPen(offscreenrp,1);
        SetDrMd(offscreenrp,JAM1);
        RectFill(offscreenrp,30,30,50,50);


        SetAPen(offscreenrp,2);
        RectFill(offscreenrp,40,40,60,60);

        SetAPen(offscreenrp,3);
        RectFill(offscreenrp,50,50,70,70);

        /* then copy it into the visible area in two different */
        /* ways... once with ClipBlit (whole rectangle comes along) */
        /* and once with BltBitMapRastPort (only colored portion */
        /* due to the choice of minterm) */
        ClipBlit(offscreenrp,30,30,rport,10,30,40,40,0xc0);

        Delay(150);

        BltBitMapRastPort(&myBitM,30,30,rport,90,30,40,40,0xc0);

        Delay(300);

cleanup:
        if(w)
           CloseWindow(w);
        if(IntuitionBase)
           CloseLibrary(IntuitionBase);
        if(GfxBase)
           CloseLibrary(GfxBase);


        for(i=0; i<DEPTH; i++)
        {
                    if(myBitM.Planes[i] != 0)
                    {
                          FreeRaster(myBitM.Planes[i],WIDTH,HEIGHT);
                    }
        }
}                         /* end of main */
```

Listing 4.17: The offscreen program (continued)

might want to develop an off-screen object with a transparent background color. Listing 4.18 implements copying with transparency. The listing uses a new function called ClipBlitTransparent and three other associated functions that are needed to allocate or delete items it needs.

ClipBlitTransparent turns the ClipBlit routine into a Bob (Blitter object) generator. Any color 0 in the source becomes transparent. The call to ClipBlitTransparent takes the form

```
ClipBlitTransparent(src_rp,src_x,src_y,dest_rp,dest_x,dest_y,
                                    size_x,size_y,shadow_rp,makeShadow);
```

A shadow mask rastport points to a bitmap containing a single bitplane the same size as the object to be blitted (moved). The bits in this bitplane are placed there by the CreateShadowRP routine. Wherever there is a bit of color other than color 0, there is a 1-bit in the shadow mask. During the blit, this mask is used to blast a hole in the destination area. Then the object can be placed into the hole. Object colors appear in the object area, background colors appear where no object colors are used.

You use CreateShadowBM and CreateShadowRP to allocate and prepare data structures for the ClipBlitTransparent routine. However, they don't actually create the shadow itself. To do this, you must set the makeShadow value to TRUE. If you have executed this routine once already, for a specific shadow bitmap, makeShadow can be set to FALSE. ClipBlitTransparent returns a value of 0 if successful; nonzero for failure.

The short code fragment here shows how ClipBlitTransparent is used. Listing 4.18 contains the complete example.

```
struct BitMap *sbm;
struct RastPort *srp;
srp = NULL;
sbm = CreateShadowBM(depth,width,height);
srp = CreateShadowRP(sbm,srp);
/* USER code should check for NULL return values for sbm, srp */
ClipBlitTransparent(sourceRP,sourceX,sourceY,
   destRP,destX,destY,
   sizeX,sizeY,
   srp,TRUE);
DeleteShadowRP(srp);
DeleteShadowBM(sbm);
```

A function that creates a bitmap for ClipBlitTransparent to use is named CreateShadowBM. This function allocates a bitmap space for a

```
    /* cliptransparent.c */


    #include "exec/types.h"
    #include "intuition/intuition.h"
    #include "exec/memory.h"

    #define WINDOWFLAGS {WINDOWSIZING|WINDOWDRAG|WINDOWDEPTH|WINDOWCLOSE}

struct NewWindow nw =
{
    100, 80,                /* start position */
    340, 110,               /* width, height */
    -1, -1,                 /* detail pen, block pen */
    0,
                            /* IDCMP flags */
    WINDOWFLAGS,
                            /* window flags */
    NULL,                   /* pointer to first user gadget */
    NULL,                   /* pointer to user checkmark */
    "ClipBlitTransparent",  /* window title */
    NULL,                   /* pointer to screen */
    NULL,                   /* pointer to superbitmap */
    60,60,640,200,          /* sizing info, min/max */
    WBENCHSCREEN            /* type of screen in which to open */
};

    struct GfxBase *GfxBase;
    struct IntuitionBase *IntuitionBase;

    extern struct RastPort *CreateShadowRP();
    extern struct BitMap *CreateShadowBM();
    extern struct Window *OpenWindow();


  main()
  {
    SHORT i;
    int x2, y2, error;

    struct RastPort testRP;
    struct BitMap testBM;


    struct RastPort saveRP;
    struct BitMap saveBM;


    struct RastPort *rp;
    struct Window *w;


    struct RastPort *imageShadowRP;
    struct BitMap *imageShadowBM;

    SHORT x,y;

    GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0);
    if (GfxBase == NULL)
    {
            printf("Unable to open graphics library\n");
            exit(1000);
    }
```

Listing 4.18: The cliptransparent program

```
IntuitionBase = (struct IntuitionBase *)
OpenLibrary("intuition.library",0);
if (IntuitionBase == NULL)
{
        CloseLibrary(GfxBase);
        printf("Unable to open intuition library\n");
        exit(1000);
}


w = OpenWindow(&nw);            /* open a window */
if(w == NULL)
      goto cleanup;
rp = w->RPort;

InitBitMap(&testBM,2,300,150);
InitBitMap(&saveBM,2,300,150);

error = FALSE;
for(i=0; i<2; i++)
{
   testBM.Planes[i]= (PLANEPTR)AllocRaster(300,150);
   if(testBM.Planes[i] == 0)
              error = TRUE;
}
if(error)
    goto cleanup;
for(i=0; i<2; i++)
{
   saveBM.Planes[i]= (PLANEPTR)AllocRaster(300,150);
   if(saveBM.Planes[i] == 0)
      error = TRUE;
}
if(error)
   goto cleanup;
InitRastPort(&testRP);
testRP.BitMap = &testBM;

InitRastPort(&saveRP);
saveRP.BitMap = &saveBM;
SetAPen(rp,1);
Move(rp,0,0);
Draw(rp,200,100);        /* in main window */
SetAPen(rp,2);
Move(rp,6,0);
Draw(rp,206,100);        /* in main window */
SetAPen(rp,3);
Move(rp,12,0);
Draw(rp,212,100);        /* in main window */

SetAPen(&testRP,2);
RectFill(&testRP,0,0,130,55);
SetAPen(&testRP,0);        /* cut a hole in center of rectangle */
RectFill(&testRP,10,10,120,45);

x = 20;
y = 10;
x2 = 2;
y2 = 1;

imageShadowBM = CreateShadowBM(2,130,55);
if(imageShadowBM==0)
          goto cleanup;
imageShadowRP = CreateShadowRP(imageShadowBM,NULL);
```

Listing 4.18: The cliptransparent program (continued)

```
    if(imageShadowRP==0)
            goto cleanup;

    /* INITIALIZATION */

    ClipBlit(rp,x,y,&saveRP,0,0, 130,55, 0xc0);        /* save back */

    /* This (TRUE) creates the shadow mask the very first time through, */
    /* so that all subsequent times we can simply USE it. */
    ClipBlitTransparent(&testRP,0,0,
                        rp,x,y,
                        130,55,
                        imageShadowRP,TRUE);
    ClipBlit(&saveRP,0,0, rp,x,y, 130,55, 0xc0);  /* restore back */


    for(i=0; i<60; i++)
    {
            ClipBlit(rp,x,y,&saveRP,0,0, 130,55, 0xc0);    /* save back */
              ClipBlitTransparent(&testRP,0,0,
                                  rp,x,y,
                                  130,55,
                                  imageShadowRP,FALSE);     /* USE shadowmask */
            Delay(5);
            ClipBlit(&saveRP,0,0, rp,x,y, 130,55, 0xc0);  /* restore back */
            x += x2;
            y += y2;
            if(y < 10 | y > 45)
            {
                    y2 = -y2;
                    x2 = -x2;
            }
      }
cleanup:
    for(i=0; i<2; i++)
    {
            if(testBM.Planes[i])
               FreeRaster(testBM.Planes[i],300,150);
            if(saveBM.Planes[i])
               FreeRaster(saveBM.Planes[i],300,150);
    }
    if(w) CloseWindow(w);
    DeleteShadowRP(imageShadowRP);
    DeleteShadowBM(imageShadowBM);

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


}       /* end of main() */

ClipBlitTransparent(srp,sx,sy,drp,dx,dy,width,height,shadowRP,makeShadow)
struct RastPort *srp, *drp, *shadowRP;
SHORT sx,sy,dx,dy;
SHORT width,height,makeShadow;
{

    /* FORM A SHADOW MASK BY OR'ING MULTIPLE TO SINGLE PLANE MASK */

    if(makeShadow)        /* need to make the shadow first time through */
```

Listing 4.18: The cliptransparent program (continued)

```
        ClipBlit(srp,sx,sy,shadowRP,0,0,width,height,0xe0);

                /* value of hex e0 means B and C + B not C + C not B, */
                /* translates to B + C (B OR C) which also */
                /* translates to "put a 1 wherever there is a 1 in */
                /* source OR destination." Allows all planes to merge */

    /* USE SHADOW MASK TO BLAST A HOLE IN THE DESTINATION AREA */

    ClipBlit(shadowRP,0,0,drp,dx,dy,width,height,0x20);

    /* FILL THE HOLE WITH SOURCE MATERIAL */

    ClipBlit(srp,sx,sy,drp,dx,dy,width,height,0xe0);
    return(0);
}

struct BitMap
*CreateShadowBM(depth,width,height)
SHORT depth, width, height;
{
        /* Shadow mask is one bitplane deep, and only as wide */
        /* and high as the area we want to move. It gets one */
        /* plane pointer worth of memory. BUT it LOOKS like */
        /* a full 5 or so planes deep. */


        SHORT i;
        struct BitMap *shadowBM;
        if((shadowBM =  (struct BitMap *)
               AllocMem(sizeof(struct BitMap),MEMF_CHIP))
               == NULL )
               return(NULL);

        InitBitMap(shadowBM,depth, width,height);

        if((shadowBM->Planes[0] = (PLANEPTR)
             AllocMem(RASSIZE(width,height),
                MEMF_CHIP | MEMF_CLEAR))== NULL)
        {
             FreeMem(shadowBM,sizeof(struct BitMap)); return(NULL);
        }

        for(i=1; i<depth; i++)
        {
                shadowBM->Planes[i] = shadowBM->Planes[0];
        }
        return(shadowBM);
} /* end of CreateShadowBM */

DeleteShadowBM(sbm)
struct BitMap *sbm;
{

        if(sbm != NULL)
        {
                if(sbm->Planes[0] != NULL)
                        FreeMem(sbm->Planes[0],
                                RASSIZE(8 *(sbm->BytesPerRow),sbm->Rows));
```

Listing 4.18: The cliptransparent program (continued)

```
                FreeMem(sbm,sizeof(struct BitMap));
        }
        return(0);
}

struct RastPort
*CreateShadowRP(shadowBM, oldshadowRP)
struct BitMap *shadowBM;
struct RastPort *oldshadowRP;
{

        struct RastPort *shadowRP;
        /* if non NULL oldshadowRP, then we are simply linking a new bitmap */
        /* into an existing data structure */
        if(oldshadowRP == NULL)
        {
        if((shadowRP = (struct RastPort *)
             AllocMem(sizeof(struct RastPort), MEMF_CHIP ))
                == NULL)
              return(NULL);
        InitRastPort(shadowRP);
        }
        else
        {
            shadowRP = oldshadowRP;         /* use old value if there is one */
        }
        /* link together the bitmap and the rastport */

        shadowRP->BitMap = shadowBM;
        return(shadowRP);
}

DeleteShadowRP(srp)
struct RastPort *srp;
{

        if(srp != NULL)
                FreeMem(srp,sizeof(struct RastPort));
        return(0);
}
```

Listing 4.18: The cliptransparent program (continued)

single plane mask of a defined width and height. (All planes are combined so that wherever there is a 1-bit in any plane there is a 1-bit in the mask.) Space is used to hold a single plane shadow of the object to allow it to be drawn transparently into a destination area by ClipBlitTransparent.

The inputs to CreateShadowBM are the depth (the number of planes in the destination area), the width (the maximum width of the object in pixels), and the height of the object. It returns a pointer to a BitMap structure if successful. That structure contains a pointer to one bit-plane of dynamically allocated memory, which can be used with CreateShadowRP to create the actual shadow of the object in this bit-plane. The function returns 0 if there was not enough memory available to do all the operations.

ClipBlitTransparent needs an area where a mask of the object can be stored. The CreateShadowRP function provides it. CreateShadowRP creates a drawing data structure that can be used with one or more bitmap-oriented objects. The call to CreateShadowRP takes the form

```
shadowRastPort = CreateShadowRP(shadowBitMap,oldShadowRP);
```

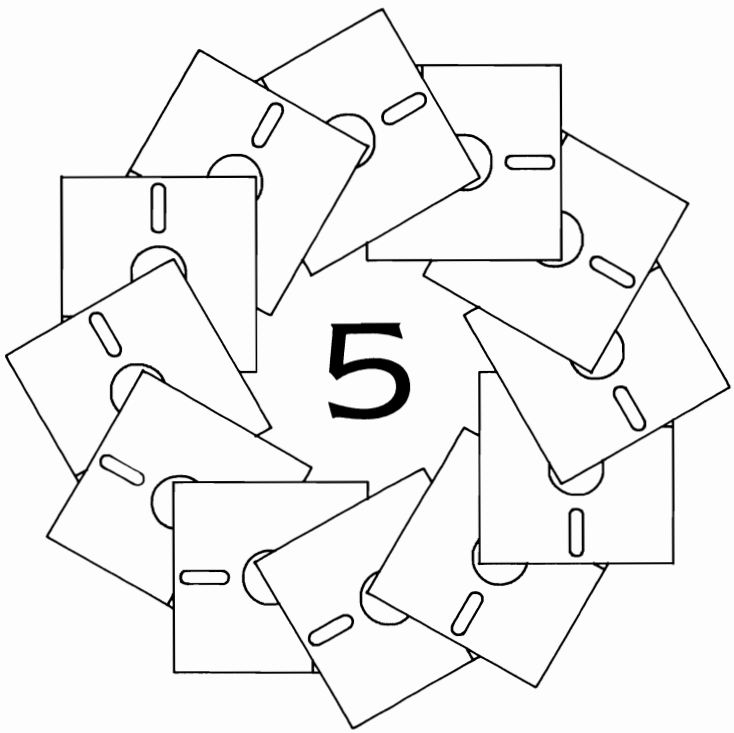
where shadowBitMap is the pointer returned from CreateShadowBM, and oldshadowRP is a pointer returned by an earlier call to CreateShadowRP. The first call must specify a value of NULL for this parameter to ask the system to dynamically allocate a RastPort data structure. The parameter is provided in lieu of a separate routine that would simply link new bitmaps into an existing rastport.

CreateShadowRP returns a pointer to a rastport that you can use to draw into the image shadow or a value of 0 if there is not enough memory.

This chapter has covered many of the Graphics library function operations of the Amiga. The graphics-rendering functions included here are all compatible with Intuition.

In the next chapter, we'll look at how Intuition constructs its displays and how it interacts with you. Along the way, we'll provide some useful basic tools that you can use to build your own Intuition applications. Onward, Intuitively...

Intuition



In this chapter you'll see a lot more interaction with Intuition than has been used thus far in the book. The Intuition screens and windows that you opened in Chapter 4 will be more thoroughly explained here.

Most of Intuition's specifications for how to draw and report things require that you specify position values relative to other things. For example, a window is positioned relative to the screen in which it appears; a requester is positioned relative to the window to which it is attached; text and gadgets are positioned relative to the requester to which they are attached; menu items are positioned relative to their menu; and subitems are positioned relative to their menu item.

This relativity gives Intuition a great deal of flexibility and saves you a lot of work later. For example, if you don't like the position at which you've placed a requester, just move the requester, and its gadgets and text go right along with it. Once you understand the relativity idea, creating windows, requesters, gadgets, and so forth becomes a lot easier.

COMMUNICATING WITH INTUITION

The most direct method for communicating with Intuition is to use the IDCMP (Intuition Direct Communications Message Port). You tell Intuition the kinds of events in which you have an interest, and every time such an event happens, Intuition sends you a message to tell you about it. The messages that you obtain from the IDCMP are called IntuiMessages.

You can get messages about the following:

Windows	You can be notified when a user activates, deactivates, sizes, or closes your window. Likewise, you can be told that your window needs refreshing (redrawing) when the system does anything that obscures or exposes part of it. You can use SIZEVERIFY to ensure that something special will be done before the system allows a resizing operation to take place. (Even though the user requests resizing, you may decide not to allow it to occur if there is not enough memory to perform a particular data move operation on which the resizing depends.)
Disks	Your software might want to know if someone has pulled out a disk that you may be planning to use or on which there is a file already open. AmigaDOS will later ask for that disk to be

reinserted if a file is indeed open. But it is useful for your software to know about the possible problem in advance.

Menus — If a user selects a menu item, the message contains the menu number, the item number within that menu, and the subitem, if any.

Gadgets — By designing a gadget, you define a rectangle either within a window or within a requester. If the user presses or releases the mouse selection button within the rectangle that is defined by one of your gadgets, your program receives a message telling which gadget has been pressed or released.

Keyboard — While yours is the active window, any input from the keyboard is routed to your window. There are two kinds of keyboard input you can request: raw key press and release sequences where you translate the keys yourself, and VANILLAKEY, where only a translated version of the key press (usually ASCII) is sent as the message. (See Chapter 6).

Timer — Intuition can generate timer events about once each one-tenth of a second. Although the Timer device (discussed in Chapter 6) is independently accessible, it may be convenient to have your task respond to this commonly available event rather than taking the trouble to set up separate communications with the Timer device.

Requesters — You can get messages when a requester has been put in place (REQSET) and when a requester has been cleared out of the drawing area (REQCLEAR). If you set REQVERIFY, you can be informed that a requester is about to be drawn and can do something, such as save the background area of a simple-refresh window, before replying. When you respond to the message, you give the system permission to draw the requester.

Mouse — Intuition keeps track of the position of the hot-spot on the mouse within the IntuitionBase structure. It also reflects the position of the mouse relative to all windows in the system as well as reporting

mouse positions within certain Intuition events. In particular, you can set the FOLLOWMOUSE flag in a Gadget structure and while the gadget is selected, your IntuiMessages will have reports of mouse movements. Or, you can set the REPORTMOUSE flag for your window and receive mouse movement reports. Or, you can set DELTAMOVE, which converts the mouse movements from absolute positions to mouse motions relative to the last move reported.

You'll find many of these features implemented in this chapter.

Messages from Intuition

Every time you get a message from Intuition, try to reply to the message as quickly as possible. Every time Intuition gets a new incoming event of one of the types you have requested, it looks to see if there are any messages that it can reuse, filling out the IntuiMessage structure with the information you have requested. If there are no messages to reuse, Intuition allocates a new empty message block for this new message and sends it to you.

If your task takes a long time to respond to messages from Intuition, Intuition continues to eat away at your free-memory area until, perhaps, you run out of available memory. Even if your task is just slow in responding, once Intuition grabs memory for use as a message block, that memory is never released until the system is rebooted. Once you allocate memory for a message, Intuition keeps the memory assigned for that message just in case it might be needed again for that same purpose. Thus, it is smart to always be sure to respond quickly and, hopefully, keep up with Intuition. This practice preserves your allocatable memory.

One method of replying quickly entails copying the IntuiMessage from Intuition's message area into your own local IntuiMessage data structure and replying immediately, before even attempting to process the message. This technique is used later in the chapter.

The Contents of an IntuiMessage

There are several data fields in an IntuiMessage structure. The most important is the Class field, which contains exactly what kind of message you are trying to interpret.

Here is a list of the contents of a typical IntuiMessage structure as defined in the Include file intuition/intuition.h:

```
struct IntuiMessage
{
    ULONG Class;
    USHORT Code;
    USHORT Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
                                /* SpecialLink is for system use only */
};
```

The Class Field

The Class field identifies the type of message that this IntuiMessage contains. The permissable values for Class are identical to the individual flags that you can set for the IDCMP messages you want to receive. When testing for Class parameter values, you can specify the same name as you used for setting the IDCMP flags.

The Code Field

The Code field is used to hold menu-related items. If the Class field contains MENUPICK, the Code field tells you which item was picked from the menu. If the Class field contains MENUVERIFY, the Code field tells you what kind of response Intuition is waiting to receive.

The Qualifier Field

The Qualifier field is simply a copy of the Qualifier field for the current input event. (IntuiMessages, in general, are copies of other system events with a few of Intuition's translations or Intuition's system variables thrown in.) The Qualifier field will have been received from the Gameport device for a mouse-related event, from the Keyboard device for a keyboard event, or from the Timer device for a timer-related event. The possible values for the Qualifier field are listed in the Include file named devices/inputevent.h.

Among the possible qualifiers are shift-key-down, left-alt-key-down, and ctrl-key-down, which enable you to further define the input event you are trying to process. For example, in a word processing program, the key combination of shift-→ might be a signal to move the cursor to the rightmost edge of the screen, whereas → alone might simply

move the cursor a single position to the right. The Qualifier field lets you make such judgements without having to keep track of the status of all such qualifier events.

The IAddress Field

The IAddress Field is for gadget-related events; it holds the address of the Gadget data structure defining the gadget that was selected. Using this address, you can get to any of the internal variables, such as the gadget identifier (GadgetID) field, and perform an operation based on this value.

MouseX and MouseY Values

These values, for all events, contain the current X and Y coordinates of the mouse relative to the upper-left corner of the screen in which the mouse is currently located. Intuition maintains the actual mouse position in an IntuitionBase structure with a possible resolution of 640 horizontal and 400 vertical positions. Due to the resolution of the cursor, it is not possible to actually place the cursor at other than 320 possible horizontal positions. The internal count is maintained at double the visible resolution to allow for future expansion and to make it possible to draw in high-resolution mode.

If the event class is a DELTAMOVE, then MouseX and MouseY specify the length of the move and the direction of the move. Positive move values indicate a move to the right, or a move down. Negative move values indicate a move to the left or a move up. DELTAMOVE values do not stop even if the mouse pointer is sitting at the edge of the screen attempting to go beyond its limits. The mouse pointer, and Intuition's internal position counter, will not proceed beyond the limits of the screen even though the DELTAMOVE values continue to increment or decrement.

If the event class is a MOUSEMOVE, then MouseX and MouseY indicate the actual position of the mouse as the event is being formulated. Note that the mouse values are valid no matter what kind of input is being reported. This is true even if the event is a MENUPICK or a GADGETDOWN or an INTUITICKS. This means that the event you receive is tied directly to the exact position of the mouse at the time the event is generated.

For example, say that you've designed a gadget that a user will select with the mouse. Not only can you get the GADGETDOWN report from Intuition, but using the MouseX and MouseY values in that event, you can tell if the user hit the top, bottom, left, right, or anywhere within that gadget. You simply have to check the mouse position against the LeftEdge, TopEdge, Width, and Height values of the Gadget structure to see where in the gadget the user clicked.

Listing 5.1 is an interpreter for that situation. Assume that the message was already received from Intuition and passed to this routine for interpretation.

As a possible alternative to making a display that has lots and lots of gadgets on it, you could, if necessary (perhaps if memory is tight), make only a single gadget (or none at all) and interpret the IntuiMessage MouseX and MouseY as shown in the listing.

Seconds and Micros Values

Each IntuiMessage structure has the Seconds and Micros fields, containing a unique time stamp for the message. No two time stamps can be alike, so this field provides a means for uniquely identifying an IntuiMessage.

The IDCMPWindow Field

This variable contains a pointer to the window that was responsible for generating this IntuiMessage. In particular, sometimes it is possible that IntuiMessages from several windows can all arrive at the same message port, perhaps common to several windows. The IDCMPWindow field allows Intuition to tag or uniquely identify the window that caused the message to be generated.

An IDCMP Message Routine

Listing 5.2 is an extended version of the event handler routine in Chapter 4. It includes all of the possible input events that an IDCMP can generate and includes making a copy of the event so as to reply quickly to Intuition. Subroutines within the main routine can be stubbed out if they are not needed. The efficient way to do this, of course, is to eliminate the case statement for each item to which you don't wish to reply. However, Listing 5.2 covers all bases with a single routine.

A few of the message classes in Listing 5.2 are mutually exclusive. That is, if you say you expect one particular message type, you prevent your task from getting any messages of another type even if you request both types of messages. The exclusions are as follows:

- If you select DELTAMOVE, you will not receive any MOUSEMOVE events. (DELTAMOVE modifies the way the mouse motion is to be reported.)
- If you select VANILLAKEY, you will not receive any RAWKEY events. (VANILLAKEY modifies the way the keyboard events are to be reported.)

```
WhereInGadget(im)
        struct IntuiMessage *im;
{
        struct Gadget *g;

        g = (struct Gadget *)im->IAddress;

        if((im->MouseX - g->LeftEdge) < 4)
                printf("Hit near left edge\n");
        else if((g->LeftEdge + g->Width-1 - im->MouseX) < 4);
                printf("Hit near right edge\n");
        else
                printf("Hit near center on X-axis\n");

        if((im->MouseY - g->TopEdge) < 4)
                printf("Hit near top edge\n");
        else if((g->LeftEdge + g->Height-1 - im->MouseY) < 4);
                printf("Hit near bottom edge\n");
        else
                printf("Hit near center on Y-axis\n");
}
```

Listing 5.1: The whereingadget function

- If you select MOUSEBUTTONS, you will not receive any GADGETDOWN or GADGETUP events for gadgets attached to windows. However, you'll still get these events for gadgets attached to requesters that are attached to windows. The mouse selection button is normally used to select gadgets. Only one event is generated per mouse button press or release. MOUSEBUTTONS takes precedence over gadget reporting.

Listing 5.3 is a file containing some routines for disabling options that are not needed for the painting program.

DESIGNING A PAINTING PROGRAM

To design a painting program, there are many things you'll have to be able to control and produce. The main goal of the simple painting program you will build in this chapter is that when the user holds down the mouse selection button, a colored dot is drawn at the position of the hot-spot on the cursor. If the mouse is moved with no buttons pressed, nothing happens.

In addition, this program allows you to call a requester that can position colored text anywhere in the drawing area.

Loading and saving pictures is not included in this program. You may want to extend the program to include these features. A close window gadget provides a means of exiting the program.

```
/* event2.c */

extern struct Window *w;

GadgetDown(m) struct IntuiMessage *m; { return(1); }

DoCloseWindow(m)
        struct IntuiMessage *m;
{
        return(FALSE);  /* let main() close the Window this time */
}

HandleEvent(ms)
        struct IntuiMessage *ms;
{
        struct IntuiMessage localms;
        int i;
        UBYTE *s, *d;
        int result;     /* result is what is returned from the */
                        /* routine and determines whether or not */
                        /* the program closes everything down */
                        /* and exits.  Return 0 if supposed to */
                        /* close and exit, nonzero if not. */
        s = (UBYTE *)ms;        /* source ... IntuiMessage */
        d = (UBYTE *)&localms;  /* dest ... local copy of IM */

        for(i=0; i< sizeof(struct IntuiMessage); i++)
        {
                *d++ = *s++;    /* copy it */
        }
        ReplyMsg(ms);           /* and reply quickly */

        /* Can use local copy to process the message contents */

        /* Response routines are written to accept the address of the */
        /* incoming IntuiMessage so that each routine can retrieve */
        /* additional information about the message if required. */
        /* If a user is really strapped for time, one can ignore */
        /* making the copy of the message (above) and instead of */
        /* passing the address of the IntuiMessage COPY, one  might */
        /* pass the address of the ORIGINAL message instead. */

        switch(localms.Class)
        {
                case SIZEVERIFY:
                        result = SizeVerify(&localms);
                        break;
                case NEWSIZE:
                        result = NewSize(&localms);
                        break;
                case REFRESHWINDOW:
                        result = RefreshWindow(&localms);
                        break;
                case MOUSEBUTTONS:
                        result = MouseButtons(&localms);
                        break;
                case MOUSEMOVE:
                        result = MouseMove(&localms);
                        break;
                case GADGETDOWN:
                        result = GadgetDown(&localms);
                        break;
                case GADGETUP:
                        result = GadgetUp(&localms);
                        break;
```

Listing 5.2: The event2 routine

```
                case REQSET:
                        result = ReqSet(&localms);
                        break;
                case MENUPICK:
                        result = MenuPick(&localms);
                        break;
                case CLOSEWINDOW:
                        result = DoCloseWindow(&localms);
                        break;
                case RAWKEY:
                        result = RawKey(&localms);
                        break;
                case REQVERIFY:
                        result = ReqVerify(&localms);
                        break;
                case REQCLEAR:
                        result = ReqClear(&localms);
                        break;
                case MENUVERIFY:
                        result = MenuVerify(&localms);
                        break;
                case NEWPREFS:
                        result = NewPrefs(&localms);
                        break;
                case DISKINSERTED:
                        result = DiskInserted(&localms);
                        break;
                case DISKREMOVED:
                        result = DiskRemoved(&localms);
                        break;
                case WBENCHMESSAGE:
                        result = WBenchMessage(&localms);
                        break;
                case ACTIVEWINDOW:
                        result = ActiveWindow(&localms);
                        break;
                case INACTIVEWINDOW:
                        result = InactiveWindow(&localms);
                        break;
                case DELTAMOVE:
                        result = DeltaMove(&localms);
                        break;
                case VANILLAKEY:
                        result = VanillaKey(&localms);
                        break;
                case INTUITICKS:
                        result = IntuiTicks(&localms);
                        break;
                default:
                        result = 1;
                        break;
                };
        return(result);

        /* As with prior versions of HandleEvent shown in */
        /* this book, the value returned from HandleEvent */
        /* will be used to determine whether or not to */
        /* close the window and end the program. */
}
```

Listing 5.2: The event2 routine (continued)

```
        /* stubs1.c */

#define IM struct IntuiMessage

        int     SizeVerify(msg)     IM *msg; { return(1); }
        int     NewSize(msg)        IM *msg; { return(1); }
        int     RefreshWindow(msg)  IM *msg; { return(1); }
        int     MouseMove(msg)      IM *msg; { return(1); }
        int     ReqSet(msg)         IM *msg; { return(1); }
        int     RawKey(msg)         IM *msg; { return(1); }
        int     VanillaKey(msg)     IM *msg; { return(1); }
        int     ReqVerify(msg)      IM *msg; { return(1); }
        int     ReqClear(msg)       IM *msg; { return(1); }
        int     MenuVerify(msg)     IM *msg; { return(1); }
        int     NewPrefs(msg)       IM *msg; { return(1); }
        int     DiskInserted(msg)   IM *msg; { return(1); }
        int     DiskRemoved(msg)    IM *msg; { return(1); }
        int     WBenchMessage(msg)  IM *msg; { return(1); }
        int     ActiveWindow(msg)   IM *msg; { return(1); }
        int     InactiveWindow(msg) IM *msg; { return(1); }
        int     DeltaMove(msg)      IM *msg; { return(1); }

        /* end of stubs1.c */
```

Listing 5.3: The stubs1 routines

Among the things you'll need to know are the following:

- How to select a screen to use
- How to determine the location of the mouse
- How to read the status of the mouse buttons
- How to get timer events for controlling the drawing
- How to design and use menus
- How to specify a color
- How to design and use gadgets

Selecting a Screen

You will want a variety of colors available to use in the painting program. The number of colors you can use depends on the depth of the screen. If you ran your painting program on the Workbench screen, you would have only four colors to choose from. In the code provided, you'll open a custom screen. Therefore, you'll have 16 possible colors from which to choose. Listing 5.4 provides the code for opening a window in a custom screen.

```
/* myscreen2.h */

struct TextAttr TestFont = { "topaz.font",8,0,0 };

struct NewScreen ns = {
0, 0,                       /* start position */
320, 200, 4,                /* width, height, depth */
0, 1,                       /* detail pen, block pen */
0,                          /* viewing mode */
CUSTOMSCREEN,               /* screen type */
&TestFont,                  /* font to use */
"Custom Screen",            /* default title for screen */
NULL };                     /* pointer to additional gadgets */


struct NewWindow nw = {
0, 10,                      /* start position */
WWIDTH, WHEIGHT,            /* width, height */
0, 1,                       /* detail pen, block pen */
INTUITICKS | GADGETUP | GADGETDOWN |
MOUSEBUTTONS | MENUPICK | CLOSEWINDOW,
                            /* IDCMP flags */
WINDOWCLOSE | ACTIVATE,     /* was BACKDROP | BORDERLESS too */
                            /* window flags */
NULL,                       /* pointer to first user gadget */
NULL,                       /* pointer to user checkmark */
"Tiny Paint",               /* window title */
NULL,                       /* pointer to screen */
NULL,                       /* pointer to superbitmap */
0,0,0,0,                    /* ignored/not a sized window so */
                            /* don't have to specify min/max */
                            /* size to allow */
CUSTOMSCREEN };             /* type of screen in which to open */


/* end of myscreen2.h */
```

Listing 5.4: The myscreen2 definition

The NewScreen Structure

In Chapter 4, screens were utilized but not discussed fully. Because creating screens is important to any Intuition programming, let's now take a closer look at the various fields in the NewScreen data structure. A NewScreen structure is defined as follows:

```
struct NewScreen
{
   SHORT LeftEdge,TopEdge,Depth,Width,Height;
   UBYTE DetailPen,BlockPen;
   USHORT ViewModes;
   USHORT Type;
   struct TextAttr *Font;
   UBYTE *DefaultTitle;
   struct Gadget *Gadgets;
   struct BitMap *CustomBitMap;
};
```

LeftEdge and TopEdge for most screens are set to 0,0. This means that the screen will be opened with its top-left corner coincident with the top-left corner of the viewing area. Height will usually be 200 lines if the viewing mode (ViewModes) is noninterlaced or 400 lines if the viewing mode is interlaced.

However, if you want to create split-screen displays, you can manipulate the screens any way you want. For example, you could open two screens, one taking up the top half of the actual viewing area (LeftEdge and TopEdge values of 0,0), and the next one taking up the lower part of the drawing area (LeftEdge and TopEdge values of 0,100). Each of the screens you create can use its own custom set of colors and any resolution you choose.

Depth specifies how many colors from which you can select for anything you draw in the screen. Depth values are 1, 2, 3, 4, or 5 for the ability to select from 2, 4, 8, 16, or 32 colors respectively.

The DetailPen is the pen number that is used to render the text for the screen title. The BlockPen is used with the DetailPen to render screen depth-arrangement gadgets.

ViewModes are a bit tricky. In this book, we use only low resolution (a value of 0 in ViewModes), high resolution (HIRES), and interlaced (LACE). In low resolution, you can define a display of up to 354 pixels across the screen. In high resolution, you can define a display up to 704 pixels. Low resolution also defines a display that is nominally 200 lines high (though you specify the value yourself). If you specify LACE as part of ViewModes, you can ask for a screen that contains up to about 400 lines within the same viewing space. This makes for a higher vertical resolution, but on some patterns causes an unacceptable level of flicker.

The Type field describes whether this is to be a WBENCHSCREEN (just like the Workbench) or a CUSTOMSCREEN. There is usually only one Workbench screen in the system. Note that the Type field in the NewWindow data structure has a real effect on how certain data fields are used.

The Font field describes the text font that should be used as a default for all menus, titles, and anything else for which the screen uses text.

DefaultTitle is a pointer to a null-terminated string that provides the title that is seen when the screen title bar is visible. The Gadgets pointer is no longer valid—screens do not get custom gadgets. It takes up space in the data structure, that's all.

The CustomBitMap pointer allows you to allocate and define your own special memory area that the system is to use to render your screen. If you do not define a custom bitmap, the system will automatically allocate the memory needed to render the screen. When the

screen closes down, this allocated memory is automatically returned to the system. For the examples in this book, this parameter is always specified as null.

The NewWindow Structure

Another important Intuition structure is NewWindow. The NewWindow data structure is defined as follows:

```
struct NewWindow
{
   SHORT LeftEdge,TopEdge;
   SHORT Width,Height;
   UBYTE DetailPen,BlockPen;
   ULONG IDCMPFlags;
   USHORT Flags;
   struct Gadget *FirstGadget;
   struct Image *CheckMark;
   UBYTE *Title;
   struct Screen *Screen;
   struct BitMap *BitMap;
   SHORT MinWidth,MinHeight;
   SHORT MaxWidth,MaxHeight;
   USHORT Type;
};
```

LeftEdge and TopEdge say where to put the window relative to the upper-left corner of the screen. Width and Height say how large to make the window, including the border and title bar. Title specifies the title that should appear in the title bar. As with screens, the DetailPen and BlockPen specify how the title bar and window gadgets (the close gadget, the depth-arrangement gadget, and the sizing gadget) should be drawn.

The IDCMP Flags are discussed in Chapter 4.

The Flags parameter specifies what kinds of gadgets are to be attached to the window when it is drawn and other characteristics of the window, as follows:

WINDOWSIZING	If this flag is set, the window can be resized. You must also specify the allowable limits on resizing in the MinWidth, MaxWidth, MinHeight, and MaxHeight variables. Caution: Width must be a value between MinWidth and MaxWidth; Height

	must be a value between MinHeight and MaxHeight.
WINDOWDRAG	If this flag is set, the window can be repositioned.
WINDOWCLOSE	If this flag is set, the window has a close gadget. Note: If you want to receive messages about the user hitting the close gadget, you must also have CLOSEWINDOW set in the IDCMP Flags parameter.
WINDOWDEPTH	If this flag is set, the window can be depth-arranged. If you have selected a BACKDROP window, when the user selects this gadget, nothing will happen.
BACKDROP	If this flag is set, the window is placed behind all other windows and never allowed to be placed in front of any other window. This conserves memory since drawing into this window uses the screen's bitmap. Windows that are opened up in front of a BACKDROP window allocate extra space for storing their own bits.
REPORTMOUSE	If this flag is set, Intuition tells you about every move the mouse makes. This is a lot of input events—you may not want to know about all of them.
GIMMEZEROZERO	The area entirely within the drawn borders of the window is considered the drawing area, starting with coordinates 0,0 in the upper-left corner. After a window has been opened, there are two sets of mouse position coordinates maintained by Intuition. The first set is MouseX, MouseY. These are the mouse positions relative to the upper-left corner of the full window, including borders. The second set of variables is GZZMouseX, GZZMouseY. This second set keeps track of the mouse if GIMMEZEROZERO is selected. It adjusts for

	the offset from the upper-left corner of the actual drawing area as compared to the actual window corner itself.
BORDERLESS	If this flag is set, there will be no borders on the window. Note that if you select WINDOWDRAG, you'll still get borders even though you did not want them. To install the dragging gadget, the system always renders the borders.
ACTIVATE	If this flag is set, this window becomes the active window when it opens. You may sometimes want to keep a different window as the active one while an informational window that is not expecting any input is opened.
RMBTRAP	If this flag is set, Right Mouse Button events are trapped from Intuition and given to your task through the IDCMP. Otherwise, all mouse menu button reports go directly to Intuition and are used to control its menus.

The following flags are mutually exclusive. You should select only one of them.

SMART_REFRESH	Intuition saves everything you draw into the window and automatically handles the refreshing (redrawing) of the window if it is obscured, then exposed.
SIMPLE_REFRESH	Intuition does not save anything drawn into obscured areas. The application itself is capable of redrawing the window. If you want to use this method, you have to ask for REFRESHWINDOW events in the IDCMP Flags.
SUPER_BITMAP	There is a superbitmap associated with this window. Superbitmap windows are discussed in Chapter 4.

The FirstGadget parameter is a pointer to the first of a set of linked gadgets that are intended to be rendered into this window. As each gadget is hit, you'll hear about them if you have GADGETDOWN and GADGETUP set in the IDCMP Flags parameter. Gadgets are covered later in this chapter.

The CheckMark parameter is a pointer to an Intuition Image data structure that describes the form of a custom checkmark image you might want to use when checking either gadgets or menu items when your window is active. If you have set a menu strip (SetMenuStrip) for your window, your menu will appear in the screen title bar when your window is active and the menu button is pressed. Within that menu, wherever there is a checkmark to be rendered, it can be your own custom checkmark. If this value is NULL, the system uses a default checkmark.

The Type parameter is critical. If you specify WBENCHSCREEN, your window will open in the Workbench screen. This is automatic. Selecting WBENCHSCREEN ignores any value in the Screen parameter in the NewWindow structure. If you specify CUSTOMSCREEN, you must have a valid pointer to a screen stored in the Screen parameter. Then, to open your window in that custom designed screen, use this sequence:

```
struct Screen *s;
struct Window *w;
s = OpenScreen(&myNewScreen);
if(s)
{
   w = OpenWindow(&myNewWindow);
   if(w)
   {
                                        /* keep on going */
   }
}
```

Finding the Mouse Location

You can find the current mouse position in the data structure for the window you opened. If w is a pointer to this window, then you could use this code:

```
CurrentMouseX = w->MouseX;
CurrentMouseY = w->MouseY;
```

However, this position will not necessarily track the moves of the mouse. Instead of the above formula, you'll want to get the mouse

position directly from the IDCMP messages. If im is a pointer to an IntuiMessage, then use this code:

```
CurrentMouseX = im->MouseX;
CurrentMouseY = im->MouseY;
```

In the painting program, if the mouse selection button is down, then each time that a timer event (INTUITICKS) comes along, a colored dot will be drawn. This is what the IntuiTicks routine in Listing 5.5 will do. A global variable named selectbutton keeps track of down (0) or up (1) status, and you act accordingly. The variables w and rp are the window pointer and the rastport pointer.

Reading the Status of Mouse Buttons

By selecting the IDCMP flag MOUSEBUTTONS in the NewWindow structure, you get messages about the status of the mouse selection button. The mouse menu button is usually reserved for Intuition to energize the menus for you. If you needed reports on the mouse menu button also, then you would specify RMBTRAP in your NewWindow flags and you would get both left and right mouse button transition reports.

Listing 5.6 is the mousebuttons subroutine. It reports the mouse selection button status.

DESIGNING AND USING MENUS

To create a pull-down menu for the painting program, you need to specify the characteristics of the menu and its items and subitems.

```
/* ticks.c */

extern int selectbutton;
int CurrentMouseX, CurrentMouseY;

extern struct RastPort *rp;

IntuiTicks(im)
        struct IntuiMessage *im;
{
        if(!selectbutton) return(1);     /* no action if it is up */

        CurrentMouseX = im->MouseX;
        CurrentMouseY = im->MouseY;

        WritePixel(rp, CurrentMouseX, CurrentMouseY);
        return(1);
}
```

Listing 5.5: The ticks routine

```
/* mousebuttons.c */

int selectbutton;       /* global - select button pressed? (T/F) */

MouseButtons(im)
struct IntuiMessage *im;
{
        if( im->Code == SELECTDOWN )
        {
                /* button just pressed */
                selectbutton = TRUE;
                IntuiTicks(im);         /* draw a pixel */
                return(1);
        }
        if( im->Code == SELECTUP)
        {
                /* button just released */
                selectbutton = FALSE;
                return(1);
        }
        return(1);      /* no handling of right button */
}
```

Listing 5.6: The mousebuttons routine

Menus appear across the top bar of the screen, replacing the screen title bar when the mouse menu button is pressed.

Each menu is fully specified by the Menu data structure. Menus are placed literally, not relatively. You specify at which horizontal and vertical position you want the left edge of a menu to begin. Each of its menu items is placed relative to the menu to which the item is attached. Thus, if the menu heading gets moved, the menu that drops down gets moved as well. If a menu item in the drop-down menu has a subitem attached to it, that subitem is positioned relative to the item to which it is attached. So if you move the menu item, its subitems move as well.

Listing 5.7 creates a menu for the painting program. There are two entries on the menu. The first is going to let the user select a color for the brush, the second is going to let the user type text into his or her picture. The program will bring up a requester that asks the user what text to add and where to put it on the picture.

How Menus and Menu Items Are Related

To better explain the program in Listing 5.7, here's how the menus and menu items relate to each other:

- Entries in a menu form a linked list that specifies a programmer-selected and spaced block of text that replaces the screen title bar when the mouse menu button is pressed and held down.

```
/* initmenu.c */

char *menunames[2];

extern struct Menu      menu[2];
struct MenuItem         textitem;
extern struct MenuItem  coloritem[32];
extern struct Image     colorimage[32];

/* value of 32 supports depth up to 5 */

InitMenu()
{
        struct Menu *menuptr;

        int n;
        int leftside;
        leftside = 2;

        menunames[0]="Color";
        menunames[1]="Text";

        menuptr = &menu[0];


        for (n=0; n<2; n++)
        {

                menuptr->LeftEdge = leftside;
                menuptr->TopEdge = 0;
                menuptr->Width = 9 * strlen(menunames[n]);
                leftside += (menuptr->Width + 2);
                menuptr->Height = 10;
                menuptr->Flags = MENUENABLED;
                menuptr->MenuName = menunames[n];

                if(n == 0)
                {
                        menuptr->FirstItem = &coloritem[0];
                        menuptr->NextMenu = &menu[1];
                }
                else
                {
                        menuptr->FirstItem = &textitem;
                        menuptr->NextMenu = NULL;
                }
        menuptr->JazzX = 0; menuptr->JazzY = 0;
        menuptr->BeatX = 0; menuptr->BeatY = 0;
        menuptr++;
        }
        InitTextItem();
        InitColorItems(DEPTH);
        return(0);
}

/* Every Menu must have at least one menu item.  Create */
/* a single item to attach to the text menu. */

struct IntuiText textitemtext =
        { 1,0,JAM2,0,0,NULL,"Insert Text",NULL };

        /* front pen, backpen, drawmode, position (left,top) */
        /* relative to (left,top) of enclosing menu rectangle, */
```

Listing 5.7: The initmenu program

```
        /* font (NULL means use the default from the Screen), */
        /* character string that is the IntuiText, link to next */
        /* IntuiText if any (this one is NULL) */

InitTextItem()
{
        textitem.NextItem = NULL;       /* only one item */

        /* When menus contain text, have to point */
        /* to IntuiText to fill them. */

        textitem.ItemFill = (APTR)&textitemtext;

        textitem.LeftEdge = 0;          /* flush left */
        textitem.TopEdge = 8;           /* next line, immediately */
                                        /* below the menu itself */
        textitem.Width = 9 * 14;        /* "Insert Text" */
                                        /*  plus space for */
                                        /*  accelerator */
        textitem.Height = 10;

        /* This MenuItem contains text, and has a shortcut */

        textitem.Flags = HIGHCOMP | ITEMTEXT | COMMSEQ | ITEMENABLED;

        /* Not trying to exclude any other selection */
        textitem.MutualExclude = 0;

        /* Nothing to render when this box is selected */
        textitem.SelectFill = NULL;

        /* Rather than use the text menu to select this item, */
        /* a user should be able to use the command-key */
        /* (left-Amiga) with a key (here a lowercase t) */
        /* to act as a command shortcut.  Your program gets */
        /* an event report just as though the menu item */
        /* were selected, without the menu ever appearing. */
        /* This works if and only if the COMMSEQ flag is */
        /* set for this MenuItem. */

        textitem.Command = 't'; /* <left-Amiga>t is shortcut */
        /* has no submenus to worry about */
        textitem.SubItem = NULL;
        textitem.NextSelect = 0;

}

/* Provide as many color images in this selection menu */
/* as there are maximum colors for given value of depth. */

InitColorItems( depth )
        SHORT depth;
{
        SHORT n,colors;
        struct Image *colorimageptr;
        struct MenuItem *coloritemptr;
        struct MenuItem *nextcoloritemptr;
        colors = palette[depth-1];

        colorimageptr = &colorimage[0];
        coloritemptr = &coloritem[0];

        nextcoloritemptr = coloritemptr;
```

Listing 5.7: The initmenu program (continued)

```
        for(n=0; n<colors; n++)
        {
                /* next... leads the item pointer by one */
                nextcoloritemptr++;

                coloritemptr->NextItem = nextcoloritemptr;
                coloritemptr->ItemFill = (APTR)colorimageptr;
                coloritemptr->LeftEdge = 2 + CW * (n % 4);
                coloritemptr->TopEdge = CH * (n / 4);
                coloritemptr->Width = CW;
                coloritemptr->Height = CH;
                coloritemptr->Flags = ITEMSTUFF;
                coloritemptr->MutualExclude = 0;

                coloritemptr->SelectFill = NULL;

                coloritemptr->Command = 0;

                coloritemptr->SubItem = NULL;
                coloritemptr->NextSelect = 0;

                /* center the image in the select box */
                colorimageptr->LeftEdge = 1;
                colorimageptr->TopEdge = 1;

                /* make the image a little smaller than the */
                /* box it is located in. */

                colorimageptr->Width = CW-2;
                colorimageptr->Height = CH-2;
                colorimageptr->Depth = depth;

                /* Not providing a bitmapped image to fit */
                /* into the box.  Instead, just taking */
                /* advantage of being able to use PlanePick */
                /* and PlaneOnOff, directly. */

                colorimageptr->ImageData = NULL;
                colorimageptr->PlanePick = 0;
                colorimageptr->PlaneOnOff = n;

                coloritemptr++; /* next one in each array */
                colorimageptr++;

                }
        coloritemptr--; /* after the loop, this pointer */
                        /* points array element beyond */
                        /* what we actually want to modify */

        coloritemptr->NextItem = NULL;  /* terminate chain */
        return(0);
}
```

Listing 5.7: The initmenu program (continued)

- In addition to the pointer to the next item within a menu, each entry of the menu also has a pointer to a menu item that drops down when that menu's active area is selected.
- The menu items form a linked list that describes the drop-down menu. Each menu item also points to an optional submenu that pops up when that menu item is selected. Only one level of sub-menu is allowed.

Initializing Menus

Menus should not overlap each other. Intuition tests the mouse position in the sequence you provide as your linked list of Menu data structures. The tests for mouse position are made as the rectangles you specify for LeftEdge, TopEdge, Width, and Height. If you create two menus that overlap, whichever is closer to the beginning of the linked list of menus takes precedence. For example, suppose you create two menus—Color and Text—and specify them as as shown in Figure 5.1. If the linkage of the menu is

```
menu[0].Next = &menu[1];
menu[1].Next = NULL;
```

then if Text is linked to menu[0], the user can select it. But if it is linked to menu[1], anytime the user is within the Color selector box, the Color menu is selected. It is not possible to select the Text menu at all. The example above uses the local variable leftside to prevent overlap.

The menu flag MENUENABLED is used for both menus. This gives the normal appearance of the menu text. If none of the menu flags is specified (the Flags parameter is set to zero), the menu cannot be selected and appears gray.

A menu must contain text, unlike menu items and subitems that can contain interesting imagery. But you can be creative and provide an interesting font to use to render that text. Notice that menu text is rendered in the same font that is used to render the screen title (and is specified in the NewScreen data structure when you open a screen). Your screen's font could contain a few funky characters, with normal text provided for the rendering of the screen title. It's up to you.

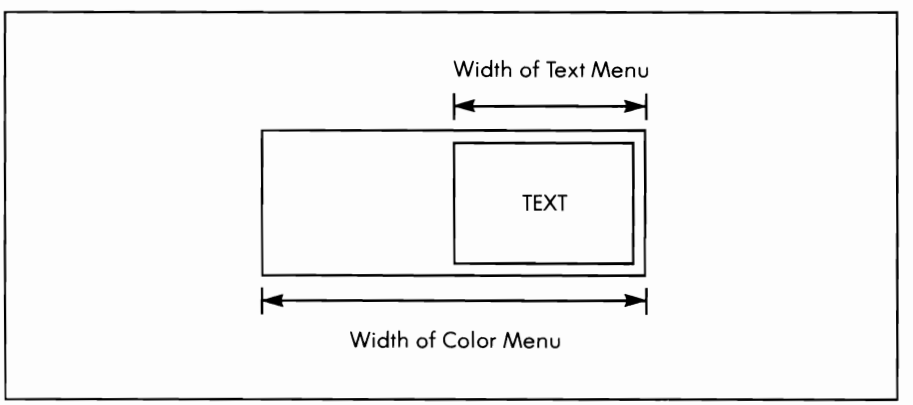


Figure 5.1: Overlapping menus

Initializing Menu Items

As with menus, you have to watch out for overlap situations as you position the various rectangles that describe menu items. With menu items, there is a more serious situation that you must watch for—lack of overlap. This may sound contradictory, but when you create a menu item list, Intuition evaluates the list of rectangular areas. On its own, Intuition creates a rectangular box that is large enough to entirely enclose all of your boxes, both the selector boxes (LeftEdge, TopEdge, Width, and Height) and the rectangular region that might be defined by the rendering of text or an image.

For example, if you specify that a menu item should be only 12 pixels wide, but then tell the system to render the word Text in Topaz font, 80-column (32 pixels wide) in that space, Intuition creates a menu item that is at least 32 pixels wide so that whatever you render will fit.

In addition, Intuition goes by some of its own rules about where a menu item can actually be positioned relative to the menu to which it is attached. The box that Intuition creates to hold your items starts at least as far left as the leftmost edge of its menu and extends at least as far right as the rightmost edge of its menu, no matter what width and positioning you request. You can ask Intuition to place an item in a out-of-the-way place, but as shown in Figure 5.2, Intuition draws the enclosing menu box by its rules.

Even if you request that the words Insert Text begin in the second position shown in the figure, Intuition still extends the enclosing box to the left so that the user can drop straight down into the menu item area. Otherwise, there would be no way to drop down and move over to the menu item. Note that the menu item cannot be selected until the mouse actually drops down, then moves across to at least the left edge of the selector box.

When you design a menu or submenu, you should design it so that there is always a small amount of overlap between adjacent items. When the user is selecting from the menu, then, at least one item is always being selected. This avoids confusing the user.

Text or Images

When you are designing menu items, you can select either text or image data to appear in them. Text takes the form of an IntuiText data structure; it requires that the MenuItem structure Flags parameter include the ITEMTEXT flag. If this flag is used, both ItemFill and SelectFill are specified as pointing to a text item.

If ITEMTEXT is not specified as one of the flags, then both ItemFill and SelectFill can point to image data. This gives you more freedom to

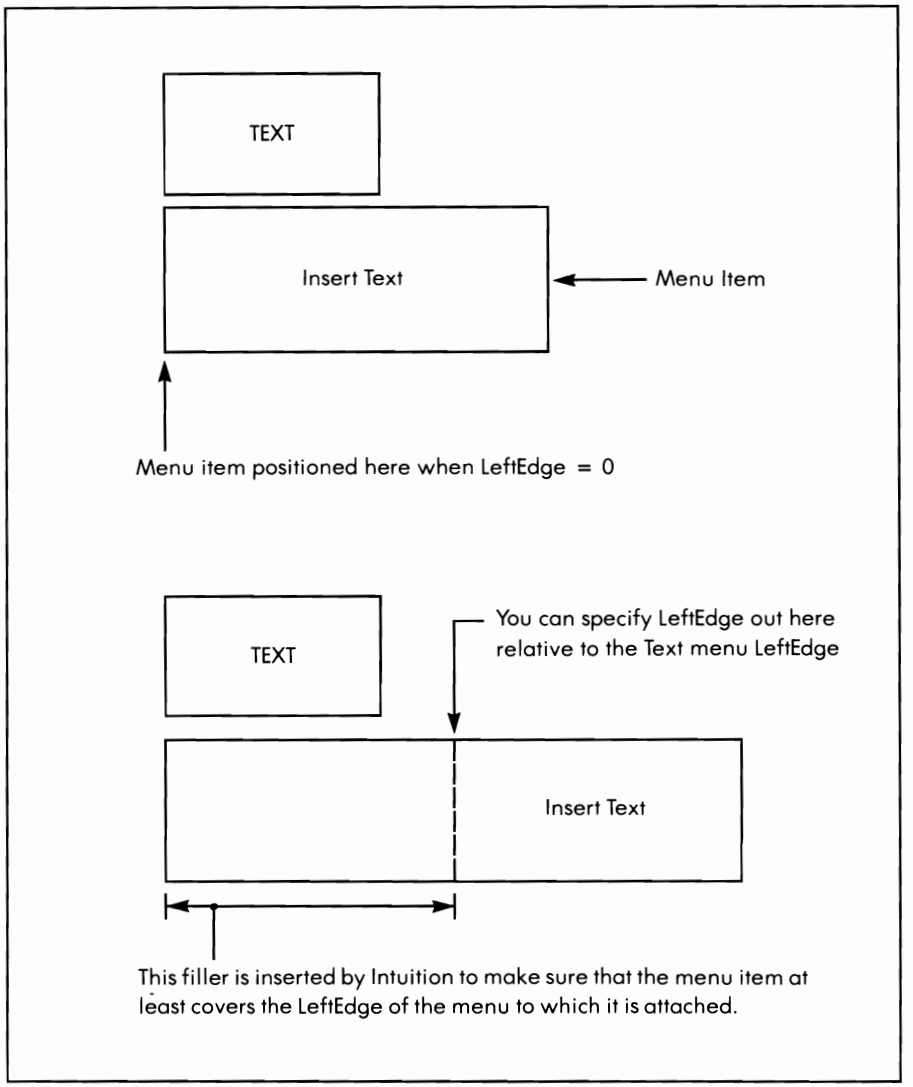


Figure 5.2: Menu-item placement

design things to put into menu items. ItemFill tells Intuition what to draw when the menu item first appears. SelectFill tells Intuition what to draw when the item is highlighted.

In Listing 5.7, the ColorItem and ColorImage data structures are initialized at the same time, since images are used exclusively in this part of the painting program. The Image data structure lets you specify where

to put the image relative to the top-left corner of its defined rectangle and the size of the image in bits wide and tall. It also lets you save memory by providing specific control over how the image is drawn (in the PlanePick and PlaneOnOff parameters).

You could, for example, design a four-color image to render in several places within a 16-color background. In several different places, perhaps, you'd like to use a different subset of the 16 available colors to render your four-color image. Intuition provides this facility, which is explained in the discussion that follows.

Color Images

Colors are formed on the screen by a binary combination of bits located in separate bitplanes for each pixel location on the screen. To select pixel color number 1 (whatever is the color currently assigned to color register number 1), use SetAPen(rp,1) and WritePixel(rp,x,y). This causes one or more bitplanes to be affected, depending on the number of bitplanes being used to make up the picture. If four bitplanes are used (giving 16 possible colors), then the combination of bits that is written at that pixel location for color number 1 is as follows:

Plane	3 2 1 0
Bit written	0 0 0 1

For color number 10:

Plane	3 2 1 0
Bit written	1 0 1 0

Say you define a four-color bit image that looks as follows:

```
1100330011
1100330011
0110330110
0011331100
0022332200
0220330220
2200330022
2200330022
```

Each of the numerical values represents a pixel of one of the possible four colors in this image. Two planes of bits combined would form the desired pattern. The bit patterns for the two planes are shown in Figure 5.3.

If you take these bits, as shown, and always draw them into the system bitplanes 0 and 1, then you'll always be mapping your bit image into system colors 0, 1, 2, and 3.

Intuition gives you another alternative: you can pick which planes your image will get rendered into. If you pick planes 3 and 1, then your image plane 0 goes into the lowest numbered plane picked (1) and your image plane 1 goes into the next lowest numbered plane picked (3).

Additionally, for the planes not picked, PlaneOnOff tells Intuition what to do with them, within the enclosing rectangle of the image. If a bit for a plane not picked is a zero, then that entire rectangle is filled with zeros. If it is a one, then everywhere within the rectangle of the image, that

Plane 0 of the image	1100110011
	1100110011
	0110110110
	0011111100
	0000110000
	0000110000
	0000110000
Plane 1 of the image	0000110000
	0000110000
	0000110000
	0000110000
	0011111100
	0110110110
	1100110011
	1100110011

Figure 5.3: A bitplane image

plane gets filled with binary ones. For the example where planes 3 and 1 are picked, planes 2 and 0 are not picked. The bits in those positions within PlaneOnOff determine what happens. Say these two bits are both ones. Here's what that looks like:

```
myImage.PlanePick = 0x0a;          /* bits 3 and 1 are picked */
myImage.PlaneOnOff = 0x05;       /* bits 2 and 0 are ON... they */
                                   /* were not picked, so they */
                                        /* 'count' this time. */
```

So, image data go into planes 3 and 1 and binary ones go into planes 2 and 0 within the defined rectangle for the image object.

The colors that result when this image is drawn are no longer 0, 1, 2, and 3, but are instead four of the available system colors. The PlaneOnOff forces plane 2 and 0 to always be ones:

Plane	3 2 1 0
Bit written	1 X 1 X

So, these are the possible colors that this image can contain:

1 0 1 0 = color 10

1 0 1 1 = color 11

1 1 1 0 = color 14

1 1 1 1 = color 15

The painting program Color menu will produce only colored rectangles, so it ignores the image data itself, saying there is no image data at all. The program specifies PlanePick = 0. That is, draw the image into no planes. This means that the color is exclusively determined by PlaneOnOff for each of the rectangles, blasting ones or zeros into the appropriate planes and thereby setting the color of the rectangle without ever specifying an image to use.

If you want to use the image capability, the format for storing the bits is as an array of unsigned words. The bit data is stored left-justified in a rectangular matrix of words. The array is wide enough to hold the width of the object and long enough to hold the height of the object. Bits left over at the rightmost edge simply remain unused. For example, if you have a 29-bit-wide object image, it takes two words (32 bits) in width to hold the image. When the image is drawn, only the leftmost 29 bits enter into the drawing considerations. The other three bits are ignored.

The image shape defined above can therefore be written as shown in Listing 5.8.

And the rest of the image would be defined as follows:

```
struct Image myImage = {
   1,1,10,8,2, &myimageshape[0],0x05,0x0a };

                                   /* LeftEdge, TopEdge, Width, Height, */
                                   /* Depth, <image>, PPick, POnOff */
```

Checkmarks

Intuition lets you add checkmarks to menu items; they appear when items are selected. Checkmarks enable a user to tell which of the options is currently active. You can make an option active by setting the CHECKIT flag in your MenuItem data structure. To use the checkmark image, be sure to leave enough space along the top-leftmost edge of the menu item into which Intuition will render the checkmark (whenever that option is selected).

The CHECKIT flag is a convenient way of having Intuition keep track of various flags for you. In your event-processing routine, you may wish to know the current state of the CHECKED flag for various menu items before you proceed. You may also want to specify the MENUTOGGLE

```
/* myimageshape.h */

SHORT myimageshape[] = {     /* plane 0 of the image */
                             0xcccc0,
                             0xcccc0,
                             0x5d80,
                             0x3f00,
                             0x0c00,
                             0x0c00,
                             0x0c00,
                             /* plane 1 of the image */
                             0x0c00,
                             0x0c00,
                             0x0c00,
                             0x0c00,
                             0x3f00,
                             0x5d80,
                             0xcccc0,
                             0xcccc0
                        };

    struct Image myImage = {
            1,1,10,8,2, &myimageshape[0],0x05,0x0a };

    /* LeftEdge, TopEdge, Width, Height, */
    /* Depth, <image>, PPick, POnOff */
```

Listing 5.8: The myimageshape routine

flag, which tells Intuition that each time the item is selected, its CHECKED state is to be reversed. That is, if it is currently CHECKED, make it unchecked; if it is not currently checked, make it CHECKED.

You can define your own image to be used for a checkmark in your NewWindow data structure. (Each window can have its own menu strip as well as its own checkmark appearance.) The CheckMark parameter in the NewWindow data structure points to NULL (if you wish to use the default checkmark) or to an Image data structure.

Why might you want to define your own checkmark? Well, the system default checkmark is about eight pixels wide by eight pixels high. You may want a different image entirely. You might also consider designing your own checkmark, perhaps a bullet (•) or an arrow, or perhaps invisible—only a single pixel wide and tall in the same color as your menu itself—so that it never appears at all. By designing your own custom invisible checkmark, you can take advantage of Intuition's CHECKED flag handling. The checkmark can be drawn or erased with no feedback to the user but you will know, because of the current state of the CHECKED flag, whether the option has been chosen. You could, for example, use the state of this flag to decide which image to present to the user the next time the menu is rendered. Also, you might want to change to alternate images for ItemFill and SelectFill based on whether the item is checked.

Mutual Exclusion

MutualExclude, in the MenuItem data structure, is a longword (32 bits) that contains one bit position for each possible menu item number. If there is a 1-bit in a particular position, then that item number is to be excluded when this item is selected.

Mutual exclusion is desireable in menus, where only one item out of a list can be chosen. Intuition allows an ideal situation in which setting the MutualExclude variable can say "if you select me, then deselect (and uncheck) the following group of selections." Thus, the exclusion of other options is fully specifiable.

Say there are these four menu items in a menu:

PATTERNFILL:

BLANK
SOLID COLOR
CROSS HATCH
STRIPED

If you use the checkmark capability, on receiving a message that any one of these menu items has been selected, you'll have to go through the entire list of menu items, reset the CHECKED flag for those items that are no longer selected, and set the CHECKED flag for the one item that is selected.

Using MutualExclude saves you this work. When any new item is selected, your task need not worry about doing anything to the rest of the items in the list. All nonselected items that are set in the MutualExclude parameter for the selected item automatically get deselected. Listing 5.9 is an example that shows how to set up the mutual exclude data for the PATTERNFILL menu. Only MutualExclude is shown, for clarity.

The system pays attention only to bits within the range of the number of menu items provided. For example, if there are only four items in a menu, it is OK for you to specify MutualExclude as having all ones except for the last four bits. The system cares about only as many bits as there are items to work with. In fact, that's what we do here—take a value that contains all 1-bits (all 1-bits will exclude all selections including this one) and reset one specific bit by using an exclusive OR. Thus, when one item is selected, all others are deselected automatically.

Highlighting Menu Items

As a user is scanning a menu with the mouse pointer, you may sometimes want to show that the system knows which item will be selected if the menu button is released. Normally, if the button is released while the

```
/* setexclude.c */

#define EXCLUDEALL 0xffffffff


SetExclude(mi,howmany)
        struct MenuItem *mi;
        int howmany;
{
        for(i=0; i<4; i++)
        {
                mi.MutualExclude =
                        EXCLUDEALL ^ (1 << i);

        /* resets only one specific bit to zero; that */
        /* bit corresponds to the identifier for this */
        /* particular menu item. */

        mi++;   /* point to the next one to set mut.ex */
        }
}
```

Listing 5.9: The setexclude routine

mouse pointer is not positioned over anything in the menu area, then no menu message gets sent to your task. Intuition gives you a choice of how to show a potential selection—by using the highlighting flags.

If you specify HIGHBOX, then, as in the listing (ITEMSTUFF = MENUENABLED + HIGHBOX), a box is drawn around each of the selector boxes as the mouse is moved.

If you specify HIGHCOMP, then the color of the box will be changed by complementing all of the colors inside the selector box. If there is text there, you'll get the effect of reverse video while the box is selected. We chose to use HIGHBOX instead of HIGHCOMP to avoid changing the colors. After all, this is for a painting program and we're trying to select a specific color.

If you specify HIGHIMAGE, it says that there is indeed a valid pointer in the MenuItem structure parameter SelectFill and that this alternate image (or text) is to be used instead of the image (or text) that is specified in ItemFill.

If you specify HIGHNONE, it leaves the user in the dark. No indication will be given by the system as to which menu item will be selected when the user releases the menu button on the mouse. Your task may get messages about menu selections made unwittingly by the user. This is not a good option to choose.

Requesters

A requester belongs to a window. It is rendered in a window, relative to the upper-left corner of that window. Requesters are implemented as separate layers of the display area. This means that even if a window is too small to hold its requester, the entire requester will still be rendered, overlapping the window's borders if necessary.

You initialize a Requester data structure by calling InitRequester and pointing to an "empty" instance of a requester. Then you specify where its upper-left corner is to be placed (LeftEdge, TopEdge) relative to the upper-left corner of its window. You also specify the requester's width and height.

A requester gets rendered by filling a rectangle with a specified color (BackFill) and drawing a border using line segments you define (ReqBorder). ReqBorder lets you draw a requester that appears to be drawn as an image floating above the background, casting a shadow against the background area. This is called a drop-shadow. Then the text (IntuiText) that you design is rendered. If you have any gadgets in this requester (there's usually at least one—OK or CANCEL), you will need to include the address of the first in a linked list of gadgets in your Requester data structure. Listing 5.10 is a requester initialization routine.

```
/* inittr.c */

struct Requester textrequest;
struct TextAttr  modfontattr;

InitTextRequest()
{
        BYTE *s, *t;
        InitRequester(&textrequest);
        textrequest.LeftEdge = 20;
        textrequest.TopEdge = 20;
        textrequest.Width = 280;
        textrequest.Height = 130;
        textrequest.ReqGadget = &trg[0];
        textrequest.ReqText = &textreqtext[0];
        textrequest.BackFill = 1;
        textrequest.ReqBorder = NULL;

        s = &textstring[0];     /* copy default string for text. */
        t = defaulttext;
        while((*s++ = *t++) != '\0')
                ;               /* do nothing but copy */


        /* set initial text mode and style */
        txfont = 80;    /* use an 80 column font */
        txmode = 1;     /* jam 1 color */

        modfontattr.ta_Name = "topaz.font";
        modfontattr.ta_YSize = 8;
        modfontattr.ta_Style = 0;
        modfontattr.ta_Flags = 0;

        return(0);
}
```

Listing 5.10: The inittr routine

Gadgets

The requester for the painting program contains gadgets for getting text, canceling an operation, and selecting the placement of text in the drawing area. This requester includes three different types of gadgets: Boolean, string, and proportional. We'll look at each of these in turn. But first, take a look at the things that are common to all types of gadgets.

Gadget Placement

If you've attached a gadget to a window, then it will be drawn relative to the upper-left corner of the window. If you've attached it to a requester, then it will be drawn relative to the upper-left corner of the requester. The placement is controlled by the LeftEdge and TopEdge parameters in the Gadget structure.

Gadget Size

The size of a gadget is defined by the Gadget structure Width and Height parameters. These parameters, as with menu items, define the

size of the hit box. That is, they define the area within which the user can click in order to select this gadget. As with menu items, the closer a gadget is to the top of its gadget list, the higher the priority it assumes with regard to user selections. If two gadgets overlap one another, it is the one that is closest to the top of the gadget list that owns the overlapped area.

Gadget Imagery

As with menu items, gadgets have one appearance when they are simply drawn, and another possible appearance when they are selected. GadgetRender and SelectRender correspond to ItemFill and SelectFill for menu items. You control the gadget highlighting by way of flags in the Gadget structure Flags parameter. As with menu items, gadgets can be complemented (GADGHCOMP), boxed (GADGHBOX), shown with the alternate image pointed to by SelectRender (GADGHIMAGE), or not highlighted at all on selection (GADGHNONE).

Unlike menu items, the imagery that SelectRender points to can be either an Image data structure or a Border data structure. You set a flag (GADGIMAGE) to tell the system that it really is an image. Border items usually take up less memory space.

Gadget Identification

When you are using menus, in general, the messages you get from Intuition are formulated by the data structure of the menus themselves. The control you have over the code number you receive for a menu selection is the placement of the menu item in its linked list. The code number is strictly position dependent.

For gadgets, when you receive notification that a gadget has been selected or released, you are given the address of the data structure that describes the gadget. Within that data structure is a field called GadgetID to which you can assign any value you choose (subject to your need for mutual exclusion.) Thus, unlike menu items that can have a maximum of only 32 items, you can use any numbering scheme you wish for your GadgetIDs, as well as having freedom to link them into a gadget list in any order you choose. Gadget hit reports are not position dependent.

Mutual Exclusion

Mutual exclusion is also available for gadgets. The MutualExclude parameter is formed in the same way as for menu items, but it depends on the GadgetID field of the Gadget structure rather than on the order in which gadgets are linked into the gadget list.

Gadget Types

Gadgets are either attached to windows or requesters. In the Gadget data structure, in the parameter field called GadgetType, the flag named REQGADGET, if set, says this is a requester gadget. Otherwise, the gadget is a window gadget. The GadgetType field contains lots of other gadget defining flags, but most of them belong to Intuition.

Gadget Flags

In addition to the flags that specify the gadget imagery, the following Gadget structure Flags are available:

GADGDISABLED	Grays out the gadget and does not allow it to be selected.
SELECTED	Renders the gadget in its selected state. You can look at this flag and find out if it is currently selected.
GRELBOTTOM	Interprets the TopEdge parameter as though it were BottomEdge. This allows you to attach a gadget to the bottom of a window or requester. If the user resizes the window, the gadget moves right along with it, always remaining visible.
GRELRIGHT	Interprets the LeftEdge parameter as though it were RightEdge. Again, this keeps gadgets available when the user resizes a window.

Gadget Activation

The Gadget Activation parameter tells Intuition what to do when a gadget is selected, and while the user is still holding down the mouse selection button. Here are a few details about activation:

GADGHIMMEDIATE	Tells Intuition to send you a message the moment that the user presses the selection button while within the hit box of this gadget. You must have also selected GADGETDOWN in your IDCMP Flags in order to receive this message.
RELVERIFY	Tells Intuition to send you a GADGETUP message when the user releases the selection button. But you'll only get the

GADGETUP message if the mouse pointer is still over the gadget. This means that if you simply want to receive GADGETDOWN and GADGETUP, you may have to do a little more work, since you cannot control where the user will place the pointer before releasing the selection button. Notice that GADGETUP must be set in your window's IDCMP Flags to get this message.

FOLLOWMOUSE — Tells Intuition to send you mouse position reports while this gadget is down. If you've set GADGIMMEDIATE, you know when the gadget went down. Even though you have requested mouse movements by setting FOLLOWMOUSE, you must also set the MOUSEMOVE flag in your window's IDCMP Flags. If FOLLOWMOUSE is set, then your event processing loop can watch for any mouse-related event other than a MOUSEMOVE, inferring that the selection button has finally been released.

ENDGADGET — If a gadget is installed in a requester, when a user selects such a gadget, it automatically ends the requester, as though EndRequest has been called. ENDGADGET is used in the painting program for the cancel button.

Border Flags — Four border flags are located in the Activation flags: RIGHTBORDER, TOPBORDER, BOTTOMBORDER, and LEFTBORDER. If any of these are set, that border is adjusted to make room for the gadget. These flags are used for such things as scroll bars. For a GIMMEZERO-ZERO window, you wouldn't want the graphics to overwrite the scroll bars or other such gadgets. These flags ask the system to make sure there is a protected space in which gadgets can be drawn.

Listing 5.11 is the code segment that defines all of the text requester gadgets.

Boolean Gadgets

The first gadget in Listing 5.11 is a Boolean gadget. A Boolean gadget has just two states, selected or not selected. For the Cancel gadget in the requester, the flags specify GADGHCOMP, which means "change the colors to their complements when this gadget is selected."

The Activation flags specify GADGIMMEDIATE | ENDGADGET; that is the OR'ed combination of both of these flags. When Cancel is hit, the only thing the system really needs to know is ENDGADGET. ENDGADGET makes the requester disappear automatically, so the GADGETDOWN report generated from GADGIMMEDIATE is just for convenience. More than one gadget in a requester can have the ENDGADGET flag set. If you want your program to know which gadget caused the requester to vanish, set GADGIMMEDIATE as well as ENDGADGET. Otherwise, Intuition will not generate any message to your application that this gadget was hit and you won't be able to tell which selection ended the requester.

The GadgetType parameter specifies REQGADGET | BOOLGADGET, saying this is a Boolean gadget installed in a requester. There is no border descriptor and no special rendering when the gadget is selected. If there had been a need for rendering an alternate image, then the GADGHIMAGE flag would have been set.

GadgetText points to the IntuiText that contains the word Cancel. Mutual exclusion is not being used. Special information comes in for either the string or proportional gadgets. These are described in detail later on in this section.

```
/* trgadgets.c */

struct Gadget trg[] = {          /* CANCEL */
{ &trg[1],                       /* address of next gadget */
205,115,60,9,                    /* left,top,width,height of hitbox */
GADGHCOMP,                       /* flags */
GADGIMMEDIATE | ENDGADGET,
                                 /* tell me only when user releases the */
                                 /* mouse button and if over the */
                                 /* gadget at that time */
REQGADGET | BOOLGADGET,          /* is a requester, is Boolean */
NULL,                            /* BORDER descriptor */
NULL,                            /* SELECT descriptor */
&textreqtext[11],                /* Cancel */
```

Listing 5.11: The trgadgets routine

```
0,                              /* mutual exclusion */
NULL,                           /* special info */
TEXTWRITEGADGETS + 1,           /* gadget identifier, user */
NULL},                          /* user data pointer */

{ &trg[2],                      /* text mode */
190,3,40,9,
GADGHCOMP,
RELVERIFY | GADGIMMEDIATE,
REQGADGET | BOOLGADGET,
NULL,
NULL,
&textreqtext[7],
0,
NULL,
TEXTWRITEGADGETS + 2,
NULL},
{ &trg[3],                      /* text style */
190,13,80,9,
GADGHCOMP,
RELVERIFY | GADGIMMEDIATE,
REQGADGET | BOOLGADGET,
NULL,
NULL,
&textreqtext[9],
0,
NULL,
TEXTWRITEGADGETS + 3,
NULL},

{&trg[4],                       /* this is a string gadget */
55,60,140,10,                   /* left,top,width,height of hitbox */
GADGHCOMP,                      /* Flags, complement mode, needed */
                                /* as of this writing for string */
                                /* gadgets  */
RELVERIFY | ENDGADGET,          /* Activation flags, when user hits */
                                /* Return, terminates (had RELVERIFY too) */
                                /* input and deselects gadget */
REQGADGET | STRGADGET,          /* is a requester, string */
NULL,                           /* BORDER descriptor */
NULL,                           /* SELECT descriptor */
NULL,                           /* IntiuText to write there */
0,                              /* mutual exclusion (could use) */
(APTR)&textstringstuff,         /* special info */
TEXTWRITEGADGETS,               /* gadget identifier, user */
NULL},                          /* user data pointer */

                                /* a dual-proportional gadget */
{ NULL,                         /* next gadget in this list */
  190,25,                       /* left edge, top edge of hitbox */
  80,42,                        /* width and height of hitbox */
  GADGIMAGE | GADGHNONE,        /* flags */
  GADGIMMEDIATE | RELVERIFY,    /* activation flags */
  PROPGADGET | REQGADGET,       /* this is a proportional gadget */
  (APTR)&textimage,             /* render normally with this image */
  (APTR)&textimage,             /* render highlighted with this one */
  NULL,                         /* no text for this prop gadget */
  0x0,                          /* mutual exclude */
  (APTR)&textslider,            /* special info = definition of prop */
  TEXTWRITEGADGETS + 4,         /* this gadget's identifier for me */
  NULL }    };                  /* no user data on this one */
```

Listing 5.11: The trgadgets routine (continued)

All of the GadgetIDs are based on a preassigned value called TEXT-WRITEGADGETS. If you modify this program for a lot of gadgets, this is an easy way to uniquely identify the groups of gadgets.

The second and third gadgets defined in the listing are for changing the styling and rendering of the text. For the style, you can use either JAM1 or JAM2. The two ROM-resident fonts, topaz-60 and topaz-80, are the possible choices for the rendering of the text. These are Boolean gadgets also. You could, of course, install other choices using the text information in Chapter 4 as your base.

The difference between these and the Cancel gadget are that these are not ENDGADGETS. The requester remains present. What is done to process these gadgets is to provide an alternative text for the gadget and to change flag variables accordingly, thereby choosing which kind of text will be used.

String Gadgets

The fourth gadget in Listing 5.11 is a string gadget. It is used to get information from the user. The hit box is sized to show as much of a long string as you wish. We chose to accept a string only as long as the box is wide, but you could select a narrow box and a very long string.

This gadget has GADGHCOMP as a Flags parameter, which is required for string gadgets. The Activation flag has ENDGADGET set so that the requester goes away automatically if the user selects the string gadget, then presses Return. Pressing Return becomes a signal to accept the result regardless of what has been done within the string gadget (even if nothing was done).

In GadgetType, REQGADGET | STRGADGET are set. Because STRGADGET is set, Intuition knows how to interpret the SpecialInfo pointer in the Gadget structure. Specifically, Intuition knows that this is a pointer to a StringInfo data structure. The StringInfo structure is provided in Listing 5.12.

This StringInfo structure provides information above and beyond the Gadget structure itself, including a textstring parameter that tells where to put the text the user inputs into this string gadget, and a text-undo parameter that provides an undo buffer. While using the gadget, the user can implement the UNDO feature to restore the default text or the last text entered (by pressing Return).

String gadgets automatically scroll to show as many characters as the width of the gadget hit box allows. When string gadgets are drawn, they contain a data entry cursor that is visible when the string gadget is active. To control the initial appearance of the gadget, the StringInfo

```
/* stringinfostuff.c */

UBYTE textstring[10];
UBYTE textundo[10];
UBYTE *defaulttext = "test";

struct StringInfo textstringstuff = {
&textstring[0],                 /* default and final string */
&textundo[0],                   /* optional undo buffer */
0,                              /* character position in buffer */
10,                             /* max characters in buffer */
0,                              /* buffer position of first displayed */
                                /* character */
0,0,0,0,0,NULL,0 };             /* Intuition local variables */
```

Listing 5.12: The stringinfo structure

structure also defines:

- The position in the buffer at which the data entry cursor should be positioned when the gadget is first selected (0).
- The maximum number of characters this string gadget is to accept (10).
- The position within the buffer of the first character to be displayed (0).

The variables that Intuition uses to keep track of what the user is doing with this string gadget have all been set to zero.

The global variables that are shown as part of Listing 5.12 are the data storage spaces for the text string and its undo buffer. The default text is copied into the text string area before the requester is first presented. Thus, if a user selects the string gadget and presses Return, a default string value is used.

Proportional Gadgets

For demonstrating proportional gadgets, we've provided a gadget that has an arrowhead image, called a slider, which has freedom to move in both horizontal and vertical directions. To fully specify the proportional gadget, beyond the Gadget structure itself, you need three additional data structures:

- A PropInfo data structure (named textslider in Listing 5.13) that describes the enclosure for the proportional gadget as well as how the gadget behaves and its initial position.

- An Image data structure (named textimage in Listing 5.13) that describes the size of the slider and where its data can be found. This data must be in chip-accessible memory (MEMF_CHIP). If you simply used the data structure defined by textsliderimage, the slider would not be visible if your program was run on a system that had external expansion memory (beyond 512K on the Amiga 1000) installed.
- An array of unsigned words that contains the image itself. The pointer called chipsliderimage, within the actual program, is set to point to a block of allocated chip-accessible memory. The slider image is then copied into this chip memory for later use.

Listing 5.13 contains the data structures and functions that define the slider you use to position the text.

```
/* slider.c */

        /* This image contains a diamond-shape. */
        /* We'll get sneaky and use it as a */
        /* left-facing arrowhead by only */
        /* specifying use of the leftmost */
        /* 4 bits of the image. */

        UWORD textsliderimage[] = {
                                  0x03c0,
                                  0x0ff0,
                                  0x3ffc,
                                  0xffff,
                                  0x3ffc,
                                  0x0ff0,
                                  0x03c0 };

        UWORD *chipsliderimage; /* Slider image must be in chip- */
                                /* accessible memory;  the program */
                                /* allocates chip memory and then */
                                /* copies the slider image into it. */

        struct Image textimage = {
                        /* image 16 bits wide but only using left 8 */
                        /* so that left pointing arrow says where to */
                        /* put the text */
                0,0,8,7,1,&textsliderimage[0],0x1,0 };

        struct PropInfo textslider = {
                FREEHORIZ | FREEVERT,
                                /* Flags .. move freely */
                                /* in both directions */
                0,0,            /* HorizPot, VertPot .. set */
                                /* the initial horizontal, vert. */
                                /* positions, then read these */
                                /* variables while or after user */
                                /* is playing with the control */
```

Listing 5.13: The slider routine

```
                                /* to see where the knob is */
                                /* currently positioned. */
                0xffff,         /* HorizBody ... not using */
                                /* autoknob so this may not be */
                                /* necessary to set to other than 0 */
                0xffff,         /* VertBody */
                0,0,0,0,0,0,};          /* Intuition's variables */

InitChipSliderImage()
{
        int i;
        UWORD *s,*d;

        chipsliderimage = (UWORD *)AllocMem(14, MEMF_CHIP);
        if(chipsliderimage == NULL)
        {
                return(FALSE);
        }
        s = textsliderimage;    /* static data within program */
        d = chipsliderimage;    /* an area guaranteed in chip area */
        for (i=0; i<7; i++)
        {
                *d++ = *s++;    /* copy the data */
        }
        return(TRUE);
}

DeleteChipSliderImage()
{
        FreeMem(chipsliderimage, 14);
}
```

Listing 5.13: The slider routine (continued)

The Flags parameter in the PropInfo structure is set to allow both horizontal and vertical movement of the control knob (slider). The HorizPot and VertPot initial values are set to 0,0, placing the slider in the upper-left corner of the container. These values will change as the user manipulates the slider in horizontal or vertical directions. We are using static data structures here, with initial values determined at compile time. If the requester is brought up multiple times and the user moves the control slider, then each time the requester appears, the position of the slider will reflect the position last set by the user.

As the user manipulates the slider, the values in HorizPot and VertPot will vary from 0 to hexadecimal FFFF. The amount that they vary depends on the width and height of the box in which they can move. For example, if you allow 20 lines of vertical movement, that is, 20 possible vertical positions at which the user can place the slider, then there are 20 possible values that VertPot can have. Each value will differ from the adjacent value by approximately hex FFFF divided by decimal 20.

To interpret the horizontal or vertical values, use the following formula:

```
myrange = mymaxvalue - myminvalue;

                    /* determine the range of actual values that you'd like */
                                        /* the pot value to represent */

myactualvalue = myminvalue +
   ((ULONG)myrange * (ULONG)textslider.VertPot)
   / 0xFFFF;

                  /* The above is a little bit of integer arithmetic, creating */
                /* a fraction by which the "myrange" value is multiplied. */
                 /* (the multiplication precedes the division to preserve */
                                        /* as much precision as possible) */
```

Intuition also provides an autoknob capability, not used here, that lets you automatically create the slider image. The size of the automatically generated knob represents the proportion of a smaller image that is actually being used. As an example, a scroll gadget attached to a window may have an autoknob the full size of the container if the entire contents of the window are visible, or only a part of the size of the container if only part of the window contents is seen. In this case, you'd set the AUTOKNOB flag in the PropInfo structure, and not point to any slider image at all.

If you are interested in the rest of Intuition's PropInfo parameters, see the explanations in the *Amiga Intuition Manual*. The rest are primarily intended for Intuition's internal use and you needn't bother with them to be able to use the proportional gadget.

Menu Processing

To process menu selections (MENUPICKs), you use the system macros MENUNUM, ITEMNUM, and SUBNUM. These retrieve numeric values from an IntuiMessage Code parameter for which menu the selection was from (values from 0 to 30), the menu item number within that menu (values from 0 to 63), and the subitem, if any, attached to that menuitem.

It is possible that the user might play with the menus, but make no selection whatsoever. This translates to an IntuiMessage Code value of MENUNULL with an IntuiMessage Class value of MENUPICK. You can tell that this has happened by either looking at the Code value directly or by interpreting the menu, menu item, and menu subitem values individually.

In this case, MENUNUM(code) returns NOMENU, ITEMNUM(code) returns NOITEM, and SUBNUM(code) returns NOSUB. The easy way out is just to provide processing for whatever legal values you recognize, and as a default, ignore anything else.

Thus, the limits that Intuition imposes for menu operations are as follows:

- 31 textual selections within the menu strip maximum
- 63 textual or image selections within the menu item group attached to each menu selection
- 31 textual or image selections within the menu subitem group attached to each menu item selection

That's a lot of possibilities.

Listing 5.14 is the routine used to process the IntuiMessage that contains a MENUPICK event. Recall that the Color menu lets you pick a color to use for drawing, and the Text menu brings up a requester that lets you specify a position at which to place text. The menu processing does indeed take the easy way out, processing only the legal values for the IntuiMessage code, and ignoring anything else.

Gadget Event Processing

The routine in Listing 5.15 is a little bit complicated, since it is handling proportional gadgets, string gadgets, and Boolean gadgets, but a switch statement with cases for each does the trick.

Listing 5.16 includes the three routines that actually change the way text will be rendered—textmode, textstyle, and textwrite (finally render the text). The first two routines are relatively simple. The third has code that interprets the current settings of the proportional gadgets to determine where in the drawing area to place the text. You'll notice that in textstyle, the system looks directly at the values of the pointers to the identifying text. This is done in lieu of a strcmp function. If the pointers have the same value, both pointers are pointing to the same string. The IntuiText for the requester is in this module as well as the processing routines.

THE PAINTING PROGRAM

Finally, Listing 5.17 is the main body of the painting program that brings everything together.

```
/* menupick.c */

#define COLORMENU 0
#define TEXTMENU 1

#define FIRSTITEM 0

MenuPick(im)
        struct IntuiMessage *im;
{
        USHORT code, k;

        code = im->Code;

        switch(MENUNUM(code)) {

                case COLORMENU: /* set a new pen color */

                        k = ITEMNUM(code);

                        if(k >= 0 && k <= 15) /* in range? */
                        {
                                SetAPen(rp, ITEMNUM(code));
                                SetDrMd(rp, JAM1);
                        }
                        break;

                case TEXTMENU: /* bring up the requester */
                               /* in our window */

                        Request(&textrequest,w);
                        break;

                default:
                        break;
                }
                return(TRUE);
        }
```

Listing 5.14: The menupick routine

```
/* gadgetup.c */

#define GADGETID ((struct Gadget *)IAddress)->GadgetID
GadgetUp(ms)
        struct IntuiMessage *ms;
{
        SHORT id;
        struct Gadget *g;

        g = (struct Gadget *)(ms->IAddress);
        id = g->GadgetID;

        /* which gadget number was it? */

        switch(id) {
```

Listing 5.15: The gadgetup routine

```
                case TEXTWRITEGADGETS:

                        textwrite();    /* write the text! */
                        break;

                case TEXTWRITEGADGETS+1:

                        break;  /* Cancel gadget, so no action */
                                /* needed.  Besides, Cancel */
                                /* contains an ENDGADGET that */
                                /* kills the Requester. */

                case TEXTWRITEGADGETS+4:

                        break;  /* proportional gadget events. */
                                /* Don't care what happens with */
                                /* the prop gadget until the */
                                /* user actually wants to write */
                                /* the text.  THEN interpret it. */

                case TEXTWRITEGADGETS+2:

                        textstyle();    /* topaz-60, topaz-80 */
                        break;

                case TEXTWRITEGADGETS+3:

                        textmode();     /* JAM1, JAM2 */
                        break;

                default:
                        break;
                }
return(TRUE);
}
```

Listing 5.15: The gadgetup routine (continued)

```
        /* textstuff.c */

        int txfont, txmode;             /* globals for text control */

        /* forward declarations to make compiler happy */

        extern struct Window *w;
        extern struct TextAttr TestFont;
        extern struct PropInfo textslider;
        extern struct TextAttr modfontattr;
        extern struct Gadget trg[];
        extern struct Requester textrequest;

        struct IntuiText textreqtext[] = {

                /* requester IntuiText for this color requester */

                { 0,1,JAM1, 5, 3, &TestFont, "Click To Change Mode:",
```

Listing 5.16: The textstuff routines

```
            &textreqtext[1]},
        { 0,1,JAM1, 5, 13, &TestFont, "Click To Change Style:",
            &textreqtext[2]},
        { 0,1,JAM1, 5, 30, &TestFont, "Pointer Positions Text",
            &textreqtext[3]},
        { 0,1,JAM1, 5, 80, &TestFont, "Click In Text Box To Type",
            &textreqtext[4]},
        { 0,1,JAM1, 25, 90, &TestFont, "Into Drawing Area",
            &textreqtext[5]},
        { 0,1,JAM1, 5, 105, &TestFont, "Press Return To Draw Text",
            &textreqtext[6]},
        { 0,1,JAM1, 5, 115, &TestFont, "Click CANCEL To Exit",
            NULL},

        /* gadget text is not linked into above IntuiText ... */
        /* just a convenient place to store it. */

        /* gadget IntuiText for this color requester */

        { 1,0,JAM2, 1, 0, &TestFont, "JAM1",
            NULL},
        { 1,0,JAM2, 1, 0, &TestFont, "JAM2",
            NULL},
        { 1,0,JAM2, 1, 0, &TestFont, "Topaz-80",
            NULL},
        { 1,0,JAM2, 1, 0, &TestFont, "Topaz-60",
            NULL},
        { 1,0,JAM2, 1, 0, &TestFont, "CANCEL",
            NULL}
        };


textstyle()
{
        if( trg[1].GadgetText == &textreqtext[7])
        {
                trg[1].GadgetText = &textreqtext[8];
                txmode = 2;
        }
        else
        {
                if(trg[1].GadgetText == &textreqtext[8])
                {
                        trg[1].GadgetText = &textreqtext[7];
                        txmode = 1;
                }
        }
        /* we changed one of them, so refresh them all */
        RefreshGadgets(&trg[1],w,&textrequest);
}

textmode()
{
        if( trg[2].GadgetText == &textreqtext[9])
        {
                trg[2].GadgetText = &textreqtext[10];
                txfont = 60;
        }
        else
        {
                if(trg[2].GadgetText == &textreqtext[10])
                {
                        trg[2].GadgetText = &textreqtext[9];
                        txfont = 80;
```

Listing 5.16: The textstuff routines (continued)

```
                }
        }
        RefreshGadgets(&trg[1],w,&textrequest);
}

textwrite()
{
        ULONG temp1, temp2;

        struct TextFont *oldfontsave;
        struct TextFont *myfontptr;

        /* scale the positions against the actual screen size;*/
        /* same could have been done with a window and so on */

        /* use the high 8 bits of each value... good enough */
        /* resolution. */

        temp1 = ((textslider.HorizPot >> 8) * (WWIDTH-1)) >> 8;
        temp2 = ((textslider.VertPot  >> 8) * (WHEIGHT-1)) >> 8;

        /* above converts slider position to a value within the */
        /* allowable range of the window */
        Move(rp,temp1,temp2);

        if(txmode == 1)
                SetDrMd(rp,JAM1);
        else
                SetDrMd(rp,JAM2);

        if(txfont == 80)
                modfontattr.ta_YSize = 8;
        else
                modfontattr.ta_YSize = 9;

        /* save current intuition font */
        oldfontsave = rp->Font;

        /* select the font that user wants */
        myfontptr = (struct TextFont *)OpenFont(&modfontattr);

        if(myfontptr == 0)
        {
                printf("\font wont open");
                /* don't draw text if font not found */
                return(0);
        }
        SetFont(rp,myfontptr);

        /* don't draw text if font is bad */

        Text(rp,&textstring[0],strlen(textstring));

        /* restore old font */
        SetFont(rp,oldfontsave);

        /* close the new font */
        CloseFont(myfontptr);
        return(0);
}
```

Listing 5.16: The textstuff routines

```
/* main.c */

#include "exec/types.h"

#define EDITLEFT 4
#define EDITRIGHT 324     /* half a screen, means 80 pixels wide max */
#define EDITTOP 12        /* maximum 42 pixels tall for current rev. */
#define EDITBOTTOM 180
#define MAXVIEWS 9
#define BOBDEPTH 4
#define FRAMEWIDTH 80
#define FRAMEHEIGHT 42
#define getc() Read(stdin, c, 2)
#define TXHEIGHT 8

struct frame {
   SHORT xmin, ymin;
   SHORT xmax, ymax;
   struct BitMap bitmap;
   };

#define qr(r,rl,rh)  ((r >= rl && r <= rh) ? 1 : 0 )

      /* starting gadget number for gadget id's */

#define MOVEGADGETS 0x0      /* move within frame */
#define COLORGADGETS 0x10    /* for changing up to 32 colors */
            /* also includes proportional gadgets, and
             * boolean in the system colors requester */
#define TEXTCOLORGADGETS 0x30   /* for text primary color (up to 32) */
#define TEXTWRITEGADGETS 0x50   /* string, prop, and three boolean */
#define DISKRWGADGETS 0x60      /* string, bool, some error handling */

#define SCROLLGADGETS 0x68
#define RECONFIGGADGETS 0x70
#define HELPGADGETS 0x90
#define EXITGADGETS 0x98
#define DEPTH 4
#define WWIDTH 320
#define WHEIGHT 190

#include "intuition/intuition.h"
#include "exec/memory.h"

/* everything copied to RAM and compiled from there */

#include "ram:imageedit.h"
#include "ram:myscreen2.h"
#include "ram:event2.c"
#include "ram:stubs1.c"
#include "ram:ticks.c"
#include "ram:mousebuttons.c"
#include "ram:stringinfostuff.c"
#include "ram:slider.c"
#include "ram:textstuff.c"
#include "ram:trgadgets.c"
#include "ram:inittr.c"

struct Window *w;
struct RastPort *rp;
struct ViewPort *vp;
struct Screen *screen;
struct Image colorimage[32];    /* provide for max possible */
```

Listing 5.17: The main program

```
long IntuitionBase=0;
struct MenuItem coloritem[32];  /* just in case depth of 5 required */

long GfxBase=0;
struct Menu menu[2];            /* one major menu item present */

extern struct Screen *OpenScreen();
extern struct Window *OpenWindow();

#define ITEMSTUFF (ITEMENABLED | HIGHBOX)
#define CW 40   /* color block width and height for color palette */
#define CH 25

SHORT palette[]  = { 2, 4, 8, 16, 32, 64 };

#include "ram:initmenu.c"
#include "ram:menupick.c"
#include "ram:gadgetup.c"

main()
{
        struct IntuiMessage *mess;
        int havevalidimage  ;

        GfxBase = OpenLibrary("graphics.library", 0);
        if (GfxBase == NULL)
        {
                printf("Unable to open graphics library\n");
                exit(1000);
        }
        IntuitionBase = OpenLibrary("intuition.library", 0);
        if (IntuitionBase == NULL)
        {
                printf("Unable to open intuition library\n");
                exit(1000);
        }
        screen = OpenScreen(&ns);
        if (screen == NULL)
        {
                exit(1);
        }
        nw.Screen = screen;

        w = OpenWindow(&nw);            /* open a window */
        rp = w->RPort;
        vp = &w->WScreen->ViewPort;

        InitMenu();

        SetMenuStrip(w, menu);

        InitTextRequest();

        havevalidimage = FALSE;

        if(InitChipSliderImage())
        {
                textimage.ImageData = (USHORT *)chipsliderimage;
                havevalidimage = TRUE;
        }
        /* had to do InitChipSliderImage just in case system used */
        /* to run this program has more than 512K of RAM.  The */
        /* slider image must be in chip-accessible RAM. */
```

Listing 5.17: The main program (continued)

```
        while(1)        /* "forever" ... wait for close message */
        {
                /* It is possible for Intuition to send you more */
                /* than one message while your task is sleeping. */
                /* You must empty the port before putting your task */
                /* to sleep again.  The construct shown here handles */
                /* each message that it receives and only goes */
                /* to sleep when the port has been emptied. */

                mess = (struct IntuiMessage *)GetMsg(w->UserPort);

                if(mess == NULL)
                        WaitPort(w->UserPort);
                else
                        if(HandleEvent(mess) == FALSE)
                                break;
        }
        /* clear the menu strip, then close what we opened */

        if(havevalidimage)
        {
                DeleteChipSliderImage();
        }

        ClearMenuStrip(w);
        CloseWindow(w);
        CloseScreen(screen);
        CloseLibrary(GfxBase);
        CloseLibrary(IntuitionBase);

}       /* end of main() */
```

Listing 5.17: The main program (continued)

OPTIONAL EXTRAS

The painting program built in this chapter provides many useful tools for you to use in your own Intuition programs. Of course, it couldn't include every available feature. There are a couple of features included here to give you some extra ideas about things you can do to make Intuition even more versatile.

Images and Text Combined

Intuition lets you set the GADGIMAGE flag for gadgets to indicate that GadgetRender and SelectRender point to image data rather than text (IntuiText). In a similar manner, Intuition lets you set the ITEMTEXT flag for menu items to indicate that the ItemFill and SelectFill pointers point to IntuiText rather than image data. But what if you wanted your menu item or gadget to look as though it combined both image and text?

You can create this impression by simply combining two menu items or two gadgets, one containing IntuiText, the other containing image

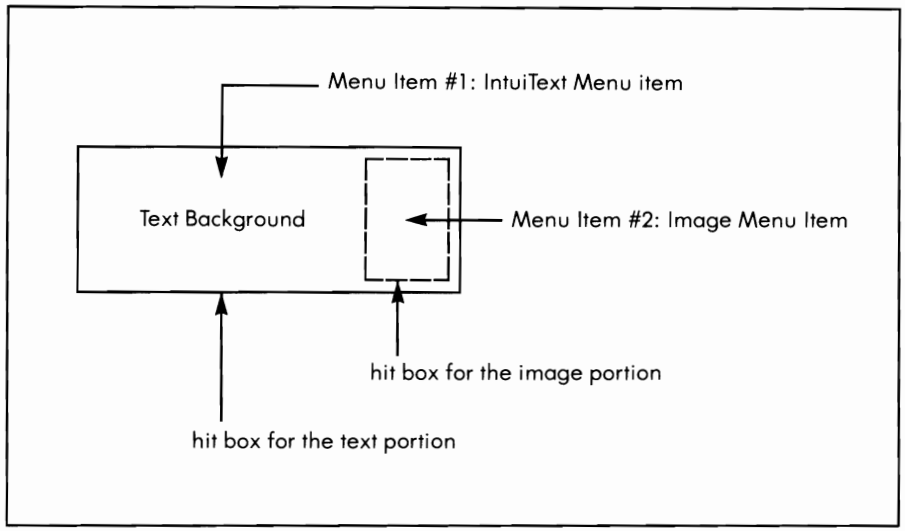


Figure 5.4: A color swatch embedded in a menu item

data. Just size the hit boxes so that one totally encloses the other. Make the outermost hit box belong to the gadget or menu item closest to the front of the linked list of gadgets or menu items. Figure 5.4 shows a color swatch (image) embedded in a menu item (IntuiText).

Menu item 1 sets the ITEMTEXT flag and is early in the menu item list. This forms the "Text Background" item. Menu item 2 does not set ITEMTEXT and is later in the list than item 1. Because item 1's hit box totally surrounds item 2, it can never be selected directly. The item itself, though composed of two distinct menu items, looks like only a single item to the user, even when selected.

Menu Item Lists

Intuition does not restrict the way in which you can use the menu item lists. Once a list is built, in addition to linking it to a menu, you can link it to one or more menu items as a submenu. For example, in the painting program, you could have built the first menu as follows:

Color

Drawing Pen

Text Foreground
Text Background

Then you could have linked the Color panel images as a subitem onto

each of the three main menu items. Here is that linkage in pseudocode:

```
<color>.FirstItem = <drawing.pen>

<drawing.pen>.NextItem = <text.foreground>
<drawing.pen>.SubItem = <first.item.in.colorimage.list>

<text.foreground>.NextItem = <text.background>
<text.foreground>.SubItem = <first.item.in.colorimage.list>

<text.background>.NextItem = NULL;
<text.background>.SubItem = <first.item.in.colorimage.list>
```

Then each one of the menu items would bring up an identical set of color panels from which the color could be chosen.

If you use this technique, you might also consider combining it with the previous technique of apparent text and image so as to show the color that is currently selected for that item. To do this, leave out the CHECKIT flag, since the subitem will be shared among three different menu items. Figure 5.5 shows how this system might appear to the user when the third item is selected and its submenu is on.

Notice that since the subitem starts at a position relative to the menu item to which it is attached, the group of color swatches will begin at precisely the same position relative to each of the three items.

Note also that you can specify a negative value for the LeftEdge and TopEdge of your menu items and still cause the object to be positioned at the appropriate place. Figure 5.6 shows how this works.

This chapter has demonstrated the primary user interface features that you'll need to be able to create Intuition programs: screens, windows, gadgets, requesters, and menus. Here are some summarizing notes:

- Screens let you define the resolution and number of colors in your drawing area.
- Windows let you split screens into multiple, overlapped areas.
- Requesters, gadgets, and menus help you to get information from users.

If you need more information about any of the topics covered in this chapter, I suggest you study the *Amiga Intuition Manual*, which describes all of the data structures and routines in great detail.

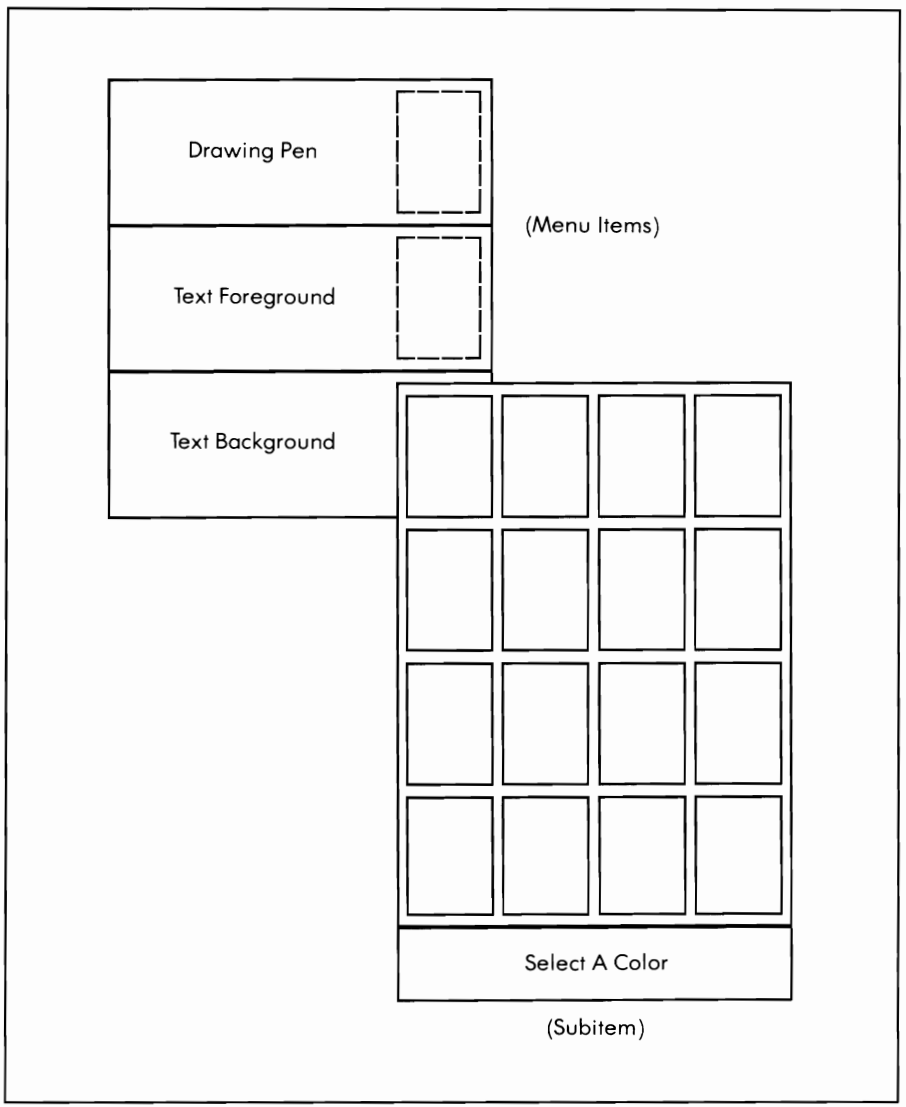


Figure 5.5: A subitem of three menu items

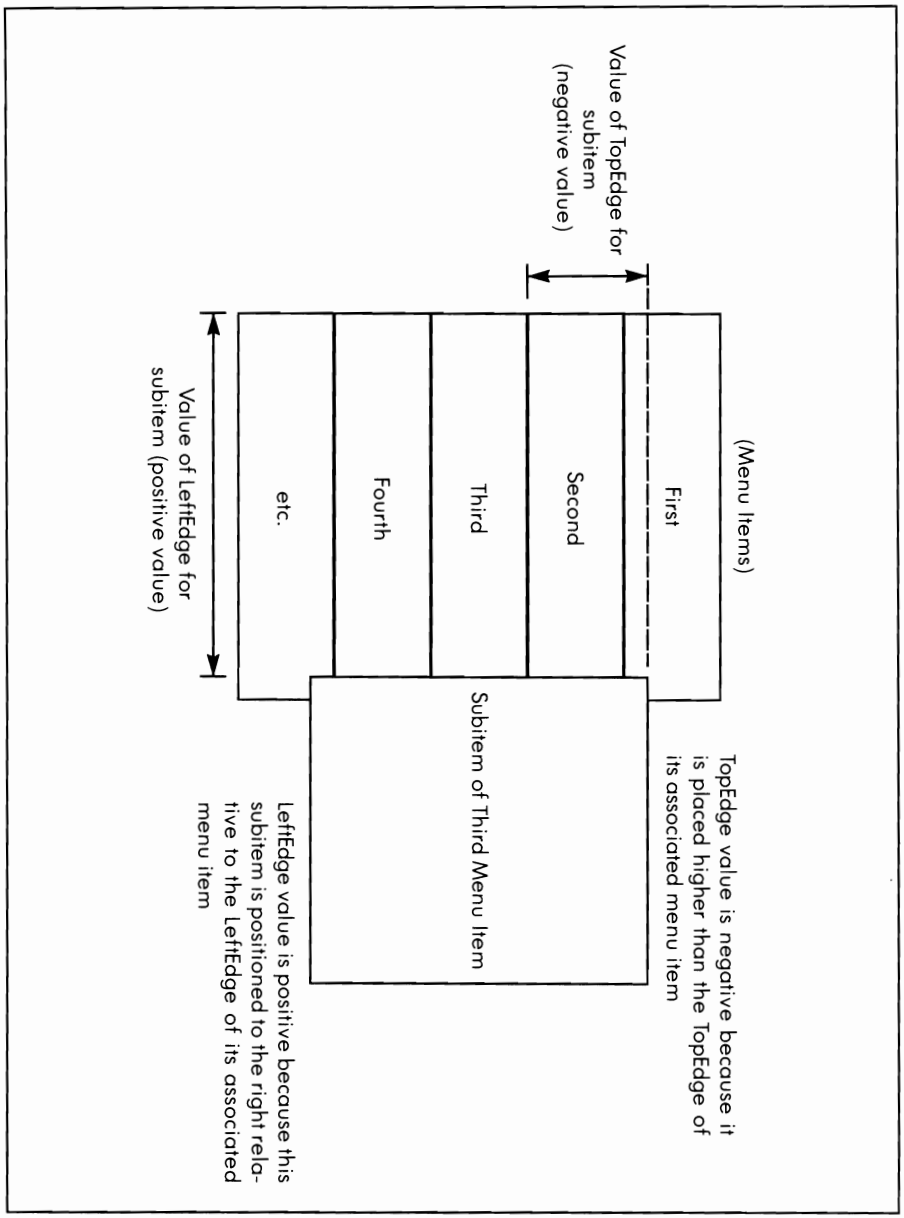
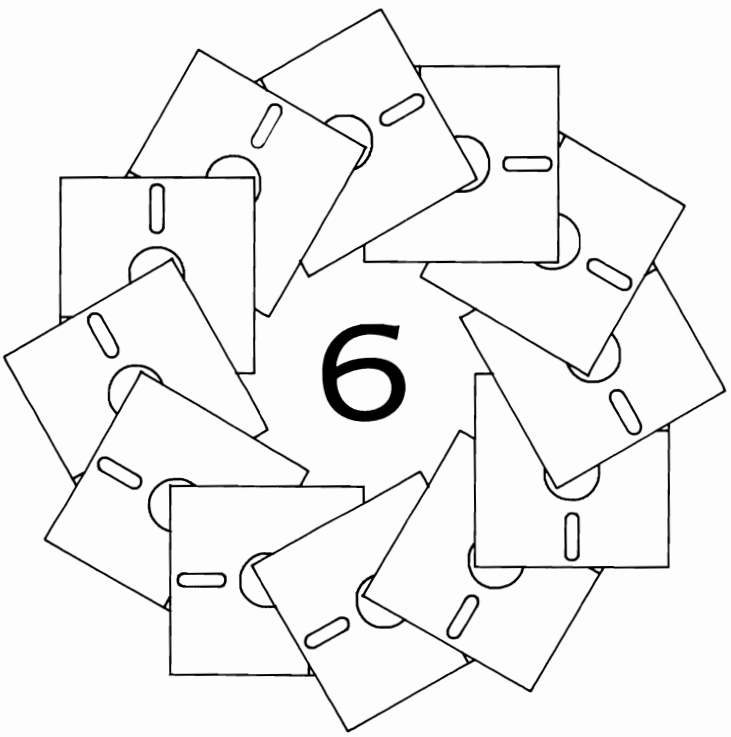


Figure 5.6: Subitem relative positioning

Devices



Recall from Chapter 3 that Exec provides a standard protocol for all I/O. I/O takes the form of a request (actually a message) that is passed from a program (task) to a device driver. The device driver takes care of all of the low-level interaction, i.e., the bit-manipulation that actually accomplishes the I/O. This means that your programs can use higher level code to communicate with devices instead of trying to control what happens at the hardware level.

A cautionary note is in order here. Once you have formulated an I/O request for a device and have sent it to the device, that block of memory you passed to the device no longer belongs to your task. Don't attempt to read or write from or to the IORequest block until the request has been completed. Many devices use the data structures contained in the IORequest block for their own purposes. You can cause a system crash if you attempt to read from or write to data structures that are not exclusively under your control.

This chapter discusses communications with the Timer, Console, Input, Keyboard, and Gameport devices. The Serial device and the Printer device are among the devices that are not discussed here. For information on these topics, see the *Amiga ROM Kernel Manual*. Note that with release 1.2 of the Amiga software, the Preferences program can be used to make the most important changes to the characteristics of the Serial device. Thus, it has become unnecessary for a program to change these characteristics.

If you tried the painting program in Chapter 5, you may have noticed that the speed of the painting was slow. In fact, the Amiga is capable of considerably faster responses than you may have noticed there.

The slow drawing speed came from the choice of when to draw another pixel. Recall that a pixel is drawn in the selected color when the mouse selection button is held down and an INTUITICKS message is received. Since these messages arrive only once each one tenth of a second, the drawing speed is limited to ten pixels per second.

There are several ways that you can improve the drawing speed. One is to set the FOLLOWMOUSE flag in the NewWindow data structure. When the selection button is held down, your program responds to mouse coordinate reports to draw the pixels. Another method is to use a faster timer.

THE TIMER DEVICE

There are two units to the Timer device. One unit manages a very accurate timer, measuring in units of 16.67 milliseconds, thereby making possible up to 60 timing events per second. That is much better than the

INTUITICKS performance, but INTUITICKS is there for convenience, not for speed. This precision timer is called the VBLANK timer.

The other timer unit is called the MICROHZ timer, since it can be programmed down to the microsecond. Rather than using this timer for its precision, though, you will more likely use it for its speed. You see, because of Amiga multitasking, a task may sleep waiting for a timer message to arrive at its message port. With other tasks going on, the task awaiting the timer message might not wake right away when the timer times out. Thus, when you set up a timer I/O request and put your task to sleep waiting for a time-out, your task will sleep at least as long as the time-out you have requested. But it might sleep for an indeterminate amount of time after that, depending on what else is going on in the system when the time-out happens.

To use the Timer device, you must do the following:

1. Select which of the timer units to use and open it
2. Create a timer IORequest block and write your timing values into it
3. Establish a reply port to receive the message from the Timer device that the time has run out
4. Write the address of this reply message port into the message block
5. Send the message block to the Timer device
6. Wait for the reply to arrive at your reply port or do something else and check for the reply later on
7. Close the timer when you are done with it

Most of the system devices can be open and accessible to only one task at a time. The Timer device, on the other hand, can service many, many tasks at once.

When you transmit a timing request to the Timer device, the device modifies the contents of the request and then sorts the list of timing requests in a relative sequence based on the request that will time-out first. In other words, if three tasks send messages to the timer, such as

- Task 1: Wake me up in 5 minutes
- Task 2: Wake me up in 1 minute
- Task 3: Wake me up in 2 minutes

the Timer device will modify the timing requests, resequence them, and change their internal meaning, as follows:

- Task 2: Wake me up 1 minute from now
- Task 3: Wake me up 1 minute later than in the previous request
- Task 1: Wake me up 3 minutes later than in the previous request

This modified set of timing requests is performed sequentially by the timer hardware. The timer request blocks are returned to each timer, in turn, after the time that they originally specified.

Each device uses its own version of an IORequest block. However, you can simplify things for the timer and utilize a standard system support function, CreateStdIO, to create a request block for the timer. In the request block that this function returns, the fields can be renamed (with a define statement) to allow you to use appropriate names for setting the time values:

```
#define SECONDS io_Actual
#define MICROSECONDS io_Length
```

Both the io_Actual and the SECONDS value in a normal timeval data structure are defined as unsigned long data, so these name assignments provide a convenient way of allocating memory for a timer request without the need to provide an additional locally defined function.

Listing 6.1 is the code you can use to create a timer request. Note that this code also creates a message port at which to receive the timer message when the timing is completed. The reply message port address is part of the parameters passed to CreatePort. You use the routine to establish an IORequest block for timer communications.

The InitTimer routine in Listing 6.2 can be used to set up the very first timing. In a program, you will do something when the timer message is received, then reinitialize the timer request block and send it off again. The timer variables timerSeconds and timerMicros are global variables so that they can be modified by other routines if necessary. To use the timerstuff routine, you must have saved the pointer to the IOStdReq block returned by PrepareTimer; you pass that pointer to this routine.

When you are finished using the timer, make sure the last time-out message has been received, then delete the timer data structures using the routine in Listing 6.3.

Listing 6.4, in pseudocode, is the sequence in which you call the routines to utilize the timer. This is an excellent chance to take advantage of

```
/* preparetimer.c */

struct IOStdReq *tr;
struct MsgPort  *tp;
struct IOStdReq *CreateStdIO();
struct MsgPort  *CreatePort();

struct IOStdReq *
PrepareTimer()
{
        tr = NULL;              /* start out unsuccessful */
        tp = CreatePort(0,0);   /* create a port with no name */
                                /* and set its priority to zero */

        if(tp == 0)
        {
                /* report the error or do something */
                /* if the port cannot be created. */

                return(tp);     /* returns NULL */
        }
        tr = CreateStdIO(tp);

        if(tr == 0)
        {
                /* again, do something if there is not enough */
                /* memory for the IOStdReq block. */

        }
        return(tp);

}       /* end of PrepareTimer() */
```

Listing 6.1: The preparetimer routine

```
/* timerstuff.c */

int timerSeconds, timerMicros;  /* globals */

InitTimer(trq)
        struct IOStdReq *trq;
{
        OpenDevice(TIMERNAME,UNIT_VBLANK,trq);
        SendTimer(trq);
        return(0);
}

SendTimer(trq)
        struct IOStdReq *trq;
{
        trq->SECONDS = timerSeconds;
        trq->MICROSECONDS = timerMicros;
        SendIO(trq);
        return(0);
}
```

Listing 6.2: The timerstuff routine

```
/* deletetimer.c */

DeleteTimer(trq)
        struct IOStdReq *trq;
{
        struct MsgPort *mp;
        mp = trq->ReplyPort;

        DeleteStdIO(trq);

        DeletePort(mp);
        return(0);
}
```

Listing 6.3: The deletetimer routine

Exec's ability to suspend execution of a task that is waiting for the occurrence of one or more events. You can tell Exec that your task wants to wake when either the timer times out or an IntuiMessage is received. (Note that you could have used UNIT_MICROHZ in place of UNIT_VBLANK in Listing 6.2.)

Recall from the main painting program in Chapter 5 that this statement can be used to put a task to sleep:

```
WaitPort(w->UserPort)
```

```
struct IOStdReq *myTimerRequest;

other program initialization...

myTimerRequest = PrepareTimer();        /* initialize a timer request */

InitTimer(myTimerRequest);              /* start the first timeout */


/* later... (loop?) */

go to sleep until timer times out then

wake up and do something until some terminating condition


then on termination:

AbortIO(myTimerRequest)  /* make sure last timer request */
                         /* has completed so we can free */
                         /* the memory resources */

DeleteTimer(myTimerRequest);
```

Listing 6.4: A timer fragment

There is another way to put a task to sleep. You can wait on a combination of signal bits where each signal bit, when set to 1, tells you that something has happened. (See Chapter 3 for a discussion of signals.) You can wait on a signal bit from either of two ports: the IDCMP (referenced as w->UserPort) and the reply port for the timer messages, which can be referenced by

```
myTimerRequest->ReplyPort
```

There is a specific signal bit associated with each port. The following define statements provide the method for accessing that particular signal bit in each type of port:

```
#define IPORTSIGNAL w->UserPort->mp_SigBit
#define TIMERSIGNAL myTimerRequest->ReplyPort->mp_SigBit
```

If you want to suspend your task until either the timer message or an IntuiMessage arrives, you use Wait instead of WaitPort and specify the combination of bits representing events that you are awaiting:

```
ULONG wakebits;                 /* provide a place to store the status of */
                                /* which bits were set when the task woke up */

wakebits = Wait(IPORTSIGNAL|TIMERSIGNAL);
```

Then, when your task awakens, you process the wake-up call in a manner similar to that shown in Listing 6.5.

```
/* multiwakeup.c */

if (wakebits & IPORTSIGNAL)
{
        /* empty the IDCMP UserPort, processing all messages */
        /* as done in event2.c in Chapter 5 */
}
if (wakebits & TIMERSIGNAL)
{
        /* if there is only one timer message sent off, then */
        /* there can be only one timer message received. */
        /* Remove the message from the ReplyPort and */
        /* reuse it to ask for the next timeout. */

        GetMsg(myTimerRequest->ReplyPort);
        timer->SECONDS = timerSeconds;
        timer->MICROSECONDS = timerMicros;
        SendIO(myTimerRequest);

        /* program will loop back to the Wait statement from */
        /* here since both types of messages (if any found) will */
        /* have been processed by the time the code gets here. */
}
```

Listing 6.5: The multiwakeup routine

As part of your final processing, you call DeleteTimer, which returns all memory resources appropriated by PrepareTimer to the system free-memory list.

THE CONSOLE DEVICE

There are many times when a programmer wants to emulate the operations of a simple ASCII terminal. The Amiga can perform terminal emulation in one or more Intuition windows by using the Console device. The Console device obeys many of the standard ANSI terminal code sequences for moving the cursor and controlling the display. In addition, there are several codes that the Console device follows that are unique to the Amiga. See Chapter 6 of the *Amiga ROM Kernel Manual* for details about exactly how the Console device handles ANSI terminal codes.

Consoles must be attached to Intuition windows. Before you create a console, you must create an Intuition window. If you attach an IDCMP to the window, the IDCMP takes precedence over the Console device regarding the messages that it receives. For example, if you specify RAWKEYS or VANILLAKEYS as one of your IDCMP flags when a window is opened, then attach a console to that window, the console will never see any keystrokes. Thus, it will never be able to interpret them for you.

Listing 6.6 provides a few supporting routines that you can include with your code to enable you to communicate effectively with the Console device. These routines provide a way to create and delete a console, and to read characters from and write characters to a console—perhaps several characters in a single function call.

The advantage to allocating memory and passing data as shown in the routines in Listing 6.6 is that each time you open a Console device, you get a unique pointer to that console's IORequest blocks (if everything went all right). Thereafter, you can refer to a specific console by using its own unique pointer to its message blocks. To remove a particular console, you use DeleteConsole, passing it the pointer to that console's message blocks.

Console Character Codes

The Console device receives its input from a number of different levels of hardware and software. The Amiga keyboard transmits a set of raw key codes—a unique code for each key, with a separate code for key presses and releases. The Keyboard device collects these keystrokes and feeds them as input events to the Input device. The Input device merges together information from the Keyboard device and the

```
/* consolestuff.c */

extern struct IOStdReq *CreateStdIO();
extern struct MsgPort *CreatePort();

struct ConIOBlocks {
        struct IOStdReq *writeReq;      /* I/O write request */
        struct IOStdReq *readReq;       /* I/O read request */
        struct MsgPort *tpr;            /* pointer to ReplyPort */
                                        /* for the console read */
};

struct ConIOBlocks *
CreateConsole(window)
        struct Window *window;
{
        struct ConIOBlocks *c;

        struct MsgPort *tpw;

        int error;

        c = (struct ConIOBlocks *)AllocMem(
            sizeof(struct ConIOBlocks), MEMF_CLEAR);

        if (c == 0)     /* out of RAM */
           goto cleanup1;

        tpw = CreatePort(0,0);  /* reply port for write */
        if (tpw == 0)
           goto cleanup2;

        c->tpr = CreatePort(0,0);  /* reply port for read */
        if (c->tpr == 0)
           goto cleanup3;

        c->writeReq = CreateStdIO(tpw);
        if(c->writeReq == 0)
           goto cleanup4;

        c->readReq = CreateStdIO(c->tpr);
        if(c->readReq == 0)
           goto cleanup5;

        c->writeReq->io_Data = (APTR)window;
        c->writeReq->io_Length = sizeof(struct Window);

        error = OpenDevice("console.device",0,c->writeReq,0);
        if(error != 0)
           goto cleanup6;       /* cannot open the console! */

        c->readReq->io_Device = c->writeReq->io_Device;
        c->readReq->io_Unit   = c->writeReq->io_Unit;
        /* Above copies the I/O request block from a */
        /* block initialized from a successful open.*/
        /* Means both read and write are talking to the */
        /* same instance of a console.   */
        return(c);      /* pointer to the ConIOBlocks */
                        /* containing both read and */
                        /* write control blocks. */
cleanup6:
        DeleteStdIO(c->readReq);
cleanup5:
        DeletePort(c->tpr);
```

Listing 6.6: The consolestuff routines

```
cleanup4:
        DeleteStdIO(c->writeReq);
cleanup3:
        DeletePort(tpw);
cleanup2:
        FreeMem(c, sizeof(struct ConIOBlocks));
cleanup1:
        return(NULL);
}

DeleteConsole(c)
        struct ConIOBlocks *c;
{
        struct MsgPort *mp;
        AbortIO(c->readReq);     /* abort any read in progress */
        CloseDevice(c->writeReq);  /* close the console device */

        mp = c->writeReq->io_Message.mn_ReplyPort;

        DeleteStdIO(c->writeReq);
        DeletePort(mp);

        mp = c->readReq->io_Message.mn_ReplyPort;

        DeleteStdIO(c->readReq);
        DeletePort(mp);

        FreeMem(c, sizeof(struct ConIOBlocks));
        return(0);
}

#define CONREAD c->readReq
#define CONWRITE c->writeReq

/* ask console, asynchronously, to read a character */

EnqueueRead(c, location)
        struct ConIOBlocks *c;
        char *location;
{
        struct IOStdReq *conr;
        conr = c->readReq;

        conr->io_Command = CMD_READ;
        conr->io_Length = 1;
        conr->io_Data = (APTR) location;
                /* buffer into which to store data read */
        SendIO(conr);
                /* asynchronous posting of a read request */
}

/* write a specified number of characters from a buffer to
 * a particular console device.
 */
WriteConsole(c, data, length)
        struct ConIOBlocks *c;
        char *data;
        WORD length;
{
        struct IOStdReq *conw;
        conw = c->writeReq;

        conw->io_Command = CMD_WRITE;   /* what to do */
```

Listing 6.6: The consolestuff routines (continued)

```
        conw->io_Length = length;       /* how many characters */
        conw->io_Data = (APTR)data;     /* where is the data? */
        DoIO(conw);             /* synchronous... wait until console */
                                /* task has accepted the data before */
                                /* going on with something else.     */

}

int
CGetCharacter(c,wait)
                /* If wait is TRUE, wait until a character        */
                /* is typed.  Return the character as the result. */
                /* If wait is FALSE, return the character value   */
                /* if a character is ready;  otherwise return -1. */

        struct ConIOBlocks *c;
        BOOL wait;
{
        struct MsgPort *mp;
        struct IOStdReq *conr;
        char *dataAddr;
        int temp;

        mp = c->tpr;    /* find the read reply port */
        if(wait)
        {
                WaitPort(mp);
        }
        conr = (struct IOStdReq *)GetMsg(mp);
        if(conr == 0)
        {
                return(-1);     /* no character present */
        }
        else
        {
                dataAddr = (char *)conr->io_Data;
                temp = *dataAddr;       /* get the value */
                EnqueueRead(c, dataAddr);  /* continue the read */
                return(temp);
        }

}       /* end of CGetChar */
```

Listing 6.6: The consolestuff routines (continued)

Gameport device, making up an input-event stream for Intuition. If Intuition does not directly interpret the events in the IDCMP, the events come through to the Console device and are translated as one or more characters per keystroke.

There is a special map that is used by the Console device to translate various key combinations. The default mapping of the Console device is such that what you see on the Amiga keyboard is pretty much what you get typed on the screen. The letter keys, for example, are translated into uppercase or lowercase ASCII characters, with the Shift key state determining the case of the letters. The number keys on the keyboard and on the numeric keypad are translated into their ASCII equivalents. The Enter key has the same effect as the Return key. Del, Tab, and Backspace transmit the expected ASCII codes.

The special keys, such as arrow keys, function keys, and the Help key transmit more than one character to the Console device for a single keystroke. The values for these special keys are listed in Table 6.1. The Control Sequence Introduction character appears as <CSI> in Table 6.1. It is a single character, having a value of hexadecimal 9B. In your input character stream from the console, a value of 9B will tell you that somebody has pressed one of these special keys.

Notice that the last two shifted values in Table 6.1 have a space between the <CSI> and the character. This space is part of the sequence transmitted.

Setting and changing console mapping is beyond the scope of this book. (Refer to the *Amiga ROM Kernel Manual*.)

Complex Input Events

You can read complex input events through the Console device. Some of the events available are raw key codes, mouse events, menu selections, gadget selections, and disk insertion and removal. You could send a special command sequence, via a console write, to the Console device to request that you receive these events. However, getting complex input events through Console device calls is not recommended. There is simply too much overhead involved. The IDCMP facility of the Intuition window can provide you with all of these events with a much lower overhead.

Raw Key Input

By setting RAWKEY as an IDCMP flag in your Intuition window, you can effectively prevent the Console device from receiving any keyboard input. You will, however, receive all of the key press and release events at your window's IDCMP, and you can translate them and respond to them in any way you wish.

Remember that RAWKEY events undergo no translation whatsoever. You get both the press and the release codes for each key the user presses on the Amiga keyboard. Figure 6.1 depicts the raw key code that each key transmits. When you receive an IntuiMessage with a message Class of RAWKEY (see Chapter 5), the message Code tells you which key was pressed. The codes shown in Figure 6.1 are for key presses. If the key code is reporting key releases, the value will be as shown with hexadecimal 80 added.

Note that the IDCMP message field named Qualifier keeps extra details about the current keyboard state, such as which Shift keys, which Amiga keys, and which Alt keys are down and whether the Control key is also down. See the Include file named devices/inputevent.h for information about the Qualifier bits.

Key	Character Codes (if Shift is up)	(if Shift is down)
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
F3	<CSI>2~	<CSI>12~
F4	<CSI>3~	<CSI>13~
F5	<CSI>4~	<CSI>14~
F6	<CSI>5~	<CSI>15~
F7	<CSI>6~	<CSI>16~
F8	<CSI>7~	<CSI>17~
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
Help	<CSI>?~	<CSI>?~
↑	<CSI>A~	<CSI>T~
↓	<CSI>B~	<CSI>S~
→	<CSI>C~	<CSI> A~
←	<CSI>D~	<CSI> @~

Table 6.1: Console device character codes for special keys

Controlling the Console Device

Instead of simply printing characters to the console, you can use certain control sequences to make the console move the cursor, erase the screen area, change how characters are output, and so on. Additionally, using OpenFont and SetFont, you can change the font that the console uses.

When you change a font in your console's window, the Console device will readjust itself to the new font. The Console device erases the drawing area, then calculates a new number of characters per line and lines per window area. The next time you output characters to the console, the new values take effect.

Controlling a console's characteristics requires that you output specific command streams to the Console device. These are listed in Table

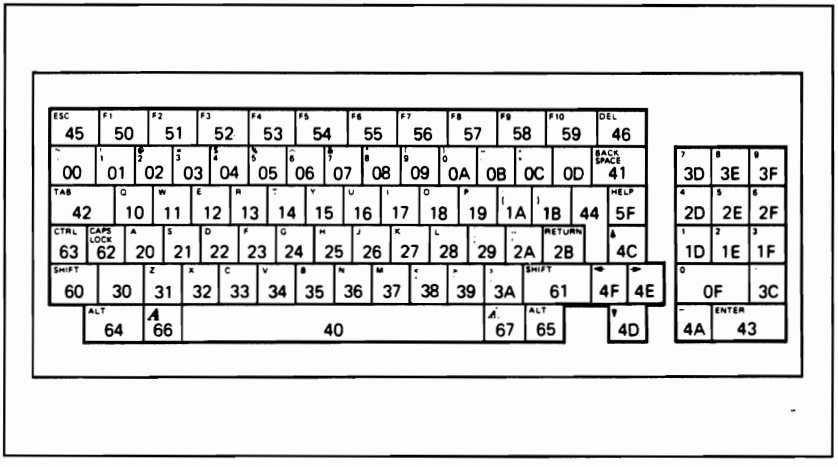


Figure 6.1: Raw key codes from the Amiga keyboard

6.2. The control-character sequences can be embedded in the output stream, right along with normal text. The sequences to transmit are listed in the table as single byte values composed of two hexadecimal digits.

In the table, the values <N> and <M> can be replaced by one or more decimal numbers. For example, insert <N> spaces, where <N> is equal to 12, would be transmitted to the Console device as byte values

9B 31 32 40

or in ASCII representation as

<CSI>12@

The <one or more values> for Select Graphic Rendition is a byte stream consisting of text style, text foreground, and text background selections. All selections are optional, but if multiple selections are made at one time, they are to be separated by a semicolon, which is a hexadecimal value 3B. Text style values (in hexadecimal) are as follows:

00	plain text
01	boldfaced
03	italicized
04	underscored
07	inverse video

Command	Byte Sequence to Transmit
Backspace (destructive)	08
Line Feed	0A
Vertical Tab	0B
Form Feed (clear console)	0C
Return	0D
Shift In (undo Shift Out)	0E
Shift Out	0F
Escape	1B
<CSI>	9B
Reset To Initial State	9B 63
Insert <N> spaces	9B <N> 40
Cursor up <N> spaces	9B <N> 41
Cursor down <N> spaces	9B <N> 42
Cursor left <N> spaces	9B <N> 43
Cursor right <N> spaces	9B <N> 44
Cursor to <N>th line following (column 1)	9B <N> 45
Cursor to <N>th line prior (column 1)	9B <N> 46
Move Cursor To Row, Column	9B <M> 3B <N> 48
Erase to end of the window	9B 4A
Erase to end of the line	9B 4B
Insert a line above this one	9B 4C
Delete this line	9B 4D

Table 6.2: Console device codes for commands

Command	Byte Sequence to Transmit
Delete <N> characters	9B <N> 50
Scroll up <N> lines	9B <N> 53
Scroll down <N> lines	9B <N> 54
Set Mode (Line Feed = Return-Line Feed)	9B 32 30 68
Reset Mode (Line Feed = Line Feed)	9B 32 30 6C
Set Page Length to <N> (recalculate <N> as the max. number of lines of the current font to try to fit within the console window)	9B <N> 74
Set Line Length to <N> (considering current font width, fit max of <N> characters on a line)	9B <N> 75
Set Left Offset to <N> (establish <N> raster columns of space at left edge of window prior to leftmost character)	9B <N> 78
Set Top Offset to <N> (leave <N> blank lines before beginning text)	9B <N> 79
Select Graphic Rendition	9B <one or more values> 6D
Device Status Report (report mouse position)	9B 6E
Window Status Report (report window size)	9B 71

Table 6.2: (continued)

Text foreground colors may be selected from the first eight colors available in the screen's colortable, with hexadecimal 30 through 37 representing color 0 through color 7 respectively. Text background colors may be selected in the same way, with hexadecimal 40 through 47 representing color 0 through color 7 respectively.

You can select multiple characteristics with a single byte stream. For example, to select boldfaced, italicized text with foreground color number 2 and background color number 3, send the following sequence to the Console device:

```
9B  01  3B  03  3B  32  3B  43  3B  6D
```

The Device Status Report command returns the current cursor position in a byte stream as

```
9B  <cursor.row>  3B  <cursor.column>  52
```

where <cursor.row> is one or more decimal digits (hex 30–39) representing the cursor row within the console window, and <cursor.column> is the column number for the cursor, also transmitted as decimal digits. The top leftmost position is (1,1).

The Window Status Report command provides information about how many rows of how many columns of text of the current font can fit into the window. The values are returned as a data stream of the form

```
9B  31  3B  31  3B  <rows>  3B  <columns>  73
```

where <rows> and <columns> are transmitted as one or more decimal digits.

Listing 6.7 uses the console subroutines previously introduced to demonstrate some of the features of the Console device, including console input, output, and control.

THE INPUT DEVICE

The Input device combines input events from both the mouse and the keyboard into a single input stream. Disk-insertion and disk-removal events are received through the Input device as well as keyboard and mouse events.

```
/* conmain.c */

/* You can create multiple consoles, each referred */
/* to by the pointer that is returned from CreateConsole */

/* EnqueueRead includes the location into which data is */
/* to be placed for the particular console */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "exec/memory.h"

ULONG IntuitionBase;

struct NewWindow nw = {
        10, 10,
        300, 100,
        -1, -1,
        0,
        GIMMEZEROZERO|ACTIVATE|SIMPLE_REFRESH|WINDOWDRAG,
        0,
        NULL,
        "Console Window",
        NULL,
        NULL,
        0,0,0,0,
        WBENCHSCREEN };

char homecursor[] = { 0x9b, '1', 0x3b, '1', 0x48 };
char backspace[]  = { 0x08 };
char linefeed[]   = { 0x0a };
char carreturn[]  = { 0x0d };
char cursorfwd[]  = { 0x9b, 0x43 };
char formfeed[]   = { 0x0c };
char insertchar[] = { 0x9b, 0x40 };
char deletechar[] = { 0x9b, 0x50 };

#define HOMECURSOR(c) WriteConsole(c,homecursor,5);
#define BACKSPACE(c)  WriteConsole(c,backspace,1);
#define LINEFEED(c)   WriteConsole(c,linefeed,1);
#define CARRETURN(c)  WriteConsole(c,carreturn,1);
#define CURSORFWD(c)  WriteConsole(c,cursorfwd,2);
#define FORMFEED(c)   WriteConsole(c,formfeed,1);
#define INSERTCHAR(c) WriteConsole(c,insertchar,2);
#define DELETECHAR(c) WriteConsole(c,deletechar,2);

/* You can add your own definitions and arrays to extend */
/* the example. */

#include "ram:console.c" /* if compiled from ram: */

main()
{
struct ConIOBlocks *cio, *CreateConsole();
struct Window *w, *OpenWindow();
int i;
char mybuffer[1];       /* where to put character */
int myinput;
char mychar[1];

IntuitionBase = OpenLibrary("intuition.library",0);
if(IntuitionBase == 0)
{
```

Listing 6.7: The conmain program

```
        printf("Intuition won't open!\n");
        exit(99);
}
w = OpenWindow(&nw);
if(w == 0)
{
        printf("Window won't open!\n");
        goto finish1;
}
cio = CreateConsole(w); /* attach a console to the window */
if(cio == 0)
{
        printf("Cannot create console!\n");
        goto finish2;
}
EnqueueRead(cio,mybuffer);
/* done ONCE, set up for read */

WriteConsole(cio, "Hello world\n\r", 13);
WriteConsole(cio, "Test backspace",14);

for(i=0; i<14; i++)
{
        BACKSPACE(cio);
        Delay(25);
}
LINEFEED(cio);
CARRETURN(cio);

WriteConsole(cio, "Test Cursor Forward", 19);
CARRETURN(cio);

for(i=0; i<19; i++)
{
        CURSORFWD(cio);
        Delay(25);
}
LINEFEED(cio);
CARRETURN(cio);

WriteConsole(cio, "Test Insert Character", 21);
CARRETURN(cio);
for(i=0; i<8; i++)
{
        INSERTCHAR(cio);
        Delay(25);
}

LINEFEED(cio);
CARRETURN(cio);

WriteConsole(cio, "********Test Delete Character", 29);
CARRETURN(cio);
for(i=0; i<8; i++)
{
        DELETECHAR(cio);
        Delay(25);
}
LINEFEED(cio);
WriteConsole(cio, "Testing Home Cursor", 19);
Delay(50);      /* wait before homing */

HOMECURSOR(cio);
```

Listing 6.7: The conmain program (continued)

```
        Delay(100);     /* 2 seconds */

        FORMFEED(cio);
        LINEFEED(cio);

        WriteConsole(cio, "Form Feed Cleared Console\n\r", 27);
        WriteConsole(cio, "Type A Line Please...\n\r\n\r", 25);
        WriteConsole(cio, "I'll echo legal characters\n\r", 28);
        WriteConsole(cio, "until you press RETURN\n\r", 24);

        /* Function keys and help key echo nothing in this mode */

        do
        {
                /* Only doing one character at a time here. */
                /* You might want to spawn a separate task */
                /* that would collect characters (type ahead) */
                /* while the console was busy doing something */
                /* else along the way. */

                myinput = CGetCharacter(cio, TRUE); /* yes, wait */
                mychar[0] = (char)(myinput & 0xff);
                WriteConsole(cio, mychar, 1);
        }
        while(mychar[0] != '\r');  /* RETURN character */

finish3:
        DeleteConsole(cio);
finish2:
        CloseWindow(w);
finish1:
        CloseLibrary(IntuitionBase);
        exit(0);
}
```

Listing 6.7: The conmain program (continued)

The Keyboard Device

The Keyboard device supplies data exclusively to the Input device. Under the current version of Exec, Intuition, and AmigaDOS, it is not possible to turn off the Input device so as to allow direct, exclusive access to the Keyboard device.

It is advisable to get keyboard input directly from Intuition's IDCMP if your task uses a window for its input or to link into the Input device and intercept keyboard messages if there is no window available.

The Gameport Device

The Gameport device also supplies data exclusively to the Input device. All mouse input events are fed to the Input device, which in turn makes them available through Intuition or at the Input device itself.

You do have a couple of choices, though. You can communicate with the Input device and assign your mouse to the second port, instead of

the default, left gameport. Also, you can communicate with the gameport and use it for joysticks and so on.

As with other devices, you must create an IORequest block to communicate with the Gameport device. You tell the device what kind of controller you have connected to the port (either a mouse or a joystick) and how to respond to the device. The program in Listing 6.8 can be compiled for either a mouse or a joystick. The default is MOUSE. To compile the program for a joystick, change the first statement from #define MOUSE 1 to #define JOYSTICK 1 and recompile.

Keyboard Enhancers

Some developers put together programs that perform keyboard remapping; some of these programs are called keyboard enhancers. To translate keyboard input into different events, you can filter all input through the Console device or through the IDCMP and translate the key sequences before they reach your application.

If, however, you are a developer trying to put together something that will work with any application rather than simply your own special application on its own custom screen, you need to go deeper into the system. In particular, you will likely want to write your own input event handler that you will install in the Input device handler chain.

There is a priority to the placement within this chain, whereby Intuition, if it gets the event first, can handle it and never pass it on to your handler. Thus, you may wish to install your handler in the chain ahead of Intuition so that you can examine the incoming events and perhaps substitute a new chain of events for your handler to pass on to Intuition.

By this method you can produce a program that can remember a user's sequence of mouse moves, selection and menu button presses, and keyboard events (in record mode). Also, you can produce a handler that can play back a recorded series of mouse and keyboard events. Or you can produce a keyboard macro enhancer that, for example, can translate a key press of F3 into "COPY myfile TO myfile.backup."

Your primary consideration is that each input event that you record takes up about 24 bytes (the size of the InputEvent data structure). Keyboard events are transmitted to the application one keystroke (press or release) at a time. Directly in an application, for example, you can reuse InputEvent memory, grabbing just one or just a few events at each I/O opportunity, and storing away only the data fields in the event in which your application is interested. For a recording-mode situation, you will most likely want to save everything about each event that comes along.

```
/*  joymouse.c */

/*  only one can be defined at a time  */

    #define MOUSE 1
/*
    #define JOYSTICK 1
*/
    #include <exec/types.h>
    #include <exec/devices.h>
    #include <graphics/gfx.h>
    #include <devices/gameport.h>
    #include <devices/inputevent.h>

    #define abs(x) (x < 0 ? -x : x)

    extern struct IOStdReq *CreateStdIO();    /* functions we use */
    extern struct MsgPort  *CreatePort();

 main()
 {
    int error,errout,delta,timeouts;
    struct GamePortTrigger trig;
    struct IOStdReq *gameMessage;          /* an I/O request */
    struct MsgPort  *gameReplyPort;        /* place to return msg */
    struct InputEvent gameEvent;           /* place to store an event */
    struct InputEvent *ge;                 /* pointer to that place */
    UBYTE *g;

    ge = &gameEvent;

    gameReplyPort = CreatePort(0,0);
    if(gameReplyPort == 0)
    {
            exit(100);      /* cannot create port */
    }
    gameMessage = CreateStdIO(gameReplyPort);
    if(gameMessage == 0)
    {
            DeletePort(gameReplyPort);
            exit(101);      /* cannot create message */
    }
    error = OpenDevice("gameport.device",1,gameMessage,0);
    /* unit 0 is left (main) port, unit 1 of gameport device is right */
    if(error)
    {
            errout = 102;
            goto cleanup;
    }
    /* now tell it what kind of controller we are using */

    gameMessage->io_Command = GPD_SETCTYPE;
    gameMessage->io_Length  = 1;           /* one byte to set type */
    gameMessage->io_Data    = (APTR)&gameEvent; /* where to find data */

    g     = (UBYTE *)&gameEvent;

#ifdef MOUSE
    *g    = (char)GPCT_MOUSE;
    delta = 5;      /* report mouse move if at least 5 clicks */
#endif MOUSE

#ifdef JOYSTICK
```

Listing 6.8: The joymouse program

```
    *g    = (char)GPCT_ABSJOYSTICK;
    delta = 1;      /* report joystick move if one click, any direction */
#endif JOYSTICK

    DoIO(gameMessage);

    if(gameMessage->io_Error)
    {
            errout = 103;
            goto cleanup; /* error while setting type */
    }
    /* now set trigger conditions, when and how does device respond? */

    gameMessage->io_Command = GPD_SETTRIGGER;
    gameMessage->io_Length  = sizeof(trig);
    gameMessage->io_Data    = (APTR)&trig;

    /* trigger a report if fire button or mouse buttons
     * are pressed or released
     */
    trig.gpt_Keys = GPTF_UPKEYS + GPTF_DOWNKEYS;

    /* trigger a report at 10 second intervals whether
     * a move, or a keypress or whatever.  (timeout in
     * vertical blanking units, 60ths of a second for
     * USA systems.)
     */
    trig.gpt_Timeout = 10 * 60;

    /* trigger a report if delta value is equal to or
     * greater than the following values
     */
    trig.gpt_XDelta = delta;
    trig.gpt_YDelta = delta;

    DoIO(gameMessage);
    if(gameMessage->io_Error)
    {
            errout = 104;
            goto cleanup;
    }
    timeouts = 0;
    gameMessage->io_Command = GPD_READEVENT;
    gameMessage->io_Data = (APTR)&gameEvent;

    do {
            gameMessage->io_Length = sizeof(struct InputEvent);

            DoIO(gameMessage);

            switch(ge->ie_Code) {

                    case (IECODE_LBUTTON):
                            printf("left-button pressed\n");
                            break;

                    case (IECODE_LBUTTON | IECODE_UP_PREFIX):
                            printf("left-button released\n");
                            break;

                    case (IECODE_RBUTTON):
                            printf("right-button pressed\n");
                            break;
```

Listing 6.8: The joymouse program (continued)

```
                case (IECODE_RBUTTON | IECODE_UP_PREFIX):
                        printf("right-button released\n");
                        break;

                case (IECODE_NOBUTTON):
                        if(abs(gameEvent.ie_X) < delta &&
                           abs(gameEvent.ie_Y) < delta)
                        {
                            printf("triggered by timeout;\n");
                            timeouts++;
                        }
                        else
                        {
                            printf("triggered by a move;\n");
                        }
#ifdef MOUSE
                        printf("mousemoves as follows:\n");
#endif MOUSE
#ifdef JOYSTICK
                        printf("joystick moves as follows:\n");
#endif JOYSTICK
                        printf("x-delta = %ld\n",
                                gameEvent.ie_X);
                        printf("y-delta = %ld\n",
                                gameEvent.ie_Y);
                default:
                        break;
                }
        }
        while(timeouts < 10);
        errout = 0;
        /* for a joystick, this program cannot tell the
         * difference between a timeout and a transition
         * from any-switch-on to no-joystick-switch-on. */

cleanup:
        DeleteStdIO(gameMessage);
        DeletePort (gameReplyPort);
        printf("Done!\n");
        exit(errout);
}       /* end of main */
```

Listing 6.8: The joymouse program (continued)

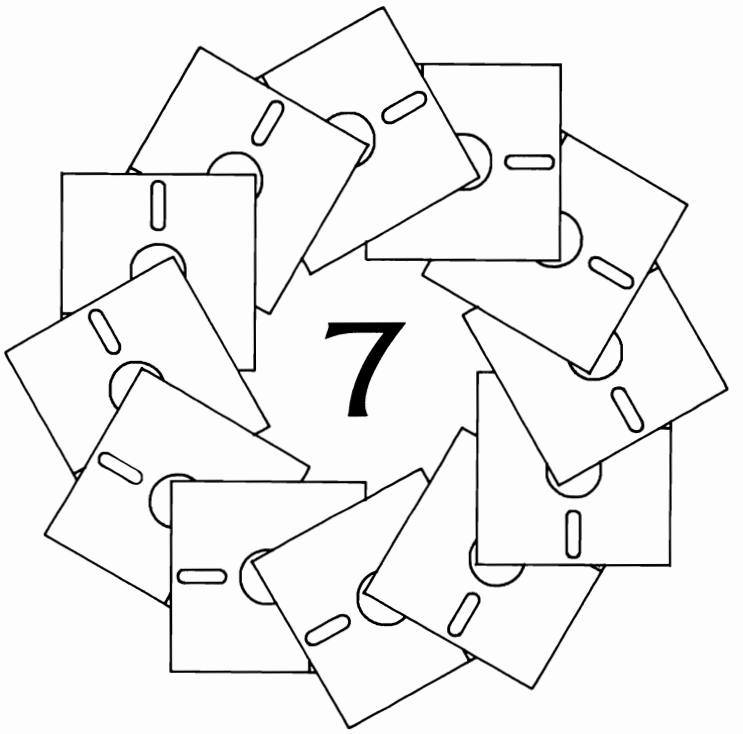
There is another interesting consideration regarding use of the Input device for keyboard event trapping. That is, what do you do if a user is switching around between several applications active on the screen (or multiple screens) at the same time? What if the user types two or three characters into one application, then activates another application and types a couple more keys?

If you are trying to create an abbreviation expander, your expander program may be unable to tell that a different application received each part of the input stream. For example, suppose you have two word-processing windows on the screen and have an abbreviation expander listening to the keyboard, with "mult" to be expanded into "multiplication". If the user types "mu" into the first window, then selects the

second window and types "It", you will want to avoid expanding the word in the second window (since the second window did not receive the complete abbreviation "mult"). You may need to tell the user that expansion takes place only for words whose abbreviation is typed entirely with no intervening mouse clicks.

Using the Input device requires not only C language, but also Amiga assembly language. Please refer to the *Amiga ROM Kernel Manual* for more information about constructing and linking Input device handlers; the manual contains a good example showing how a new handler is created and linked into the input stream.

Animation



This chapter describes the animation tools available on the Amiga: the simple sprite system and the gel (graphics element) system. The gel system consists of virtual sprites and Bobs (a short name for Blitter objects). To animate an object on the screen using the gel system, you define the object and how it moves relative to other objects on the screen. The gel system uses hardware sprites to create its virtual sprites and draws Bobs very quickly using the Blitter, a coprocessor that can move and combine data very quickly.

Objects are drawn in the sequence you define, in the positions you define, with an option to save the background area that each overlays when it is drawn. Then, when the object moves to a new location, the original background area is restored, and the objects are drawn in their new positions.

There are many options provided by the gel system, allowing you very close control over the animation process. All of the options are described here, and some are used in the example programs, which include tools that you may find useful later in trying to manipulate your own objects.

SIMPLE SPRITES

Using simple sprites, you can create and move up to seven objects, each of which is 16 low-resolution pixels wide (about one twentieth of the width of your screen) and composed of up to three different colors of your choosing. These objects can be made to move very quickly on the screen. For example, the mouse cursor is a simple sprite. Simple sprites actually use one of the underlying hardware features of the Amiga—the hardware sprite system.

The display that each sprite produces is independent of the display of the background area (called the playfield). Only in color choices is there an interaction between simple sprites and the playfield.

There may be times when you simply want to manipulate the mouse pointer, perhaps in response to a user moving a joystick or using a mouse plugged into the second mouse port. Or you might be designing a game in which you want to display small, highly mobile objects. For these purposes, you may want to use simple sprites.

The Amiga has eight simple sprites available, each related directly to the underlying sprite hardware. The difference between a simple sprite and a hardware sprite is that system software for simple sprites limits each hardware sprite to a single use within a single video display scan. It is possible, by directly manipulating the sprite hardware, to reuse the hardware sprites many times in a single display. Simple sprites, though,

do not take advantage of that capability, but instead provide an easy method to determine precisely where an object is located and how it appears to the user.

The SimpleSprite Data Structure

You designate a SimpleSprite data structure for each of the sprites you want to use in your program. This data structure is as follows (from graphics/sprite.h):

```
struct SimpleSprite
{
   UWORD *posctldata;
   UWORD height;
   UWORD x,y;
   UWORD num;
};
```

To establish the contents of most of the data fields, you use system routines, such as ChangeSprite and MoveSprite. However, you can read the contents of the SimpleSprite structure to determine the current location of the sprite object or to determine which hardware sprite has been assigned by the system so that you can directly control the sprite's colors.

Each simple sprite, when it appears on the screen, is 16 pixels wide in low-resolution mode. It does not matter whether the sprite is appearing over a low-resolution screen (320 pixels across) or a high-resolution screen (640 pixels across). Sprites always use low-resolution mode for their picture elements (pixels).

The height of the sprite is determined by the Height value stored in its SimpleSprite data structure. The position of the sprite is determined by the x and y values in the data structure. How the sprite appears is determined by the memory area to which the posctldata pointer points.

Obtaining a Sprite

Before you can build any data structures for sprites, you must first obtain a sprite from the system so that you can assign that sprite for your own use. To get a sprite, you use the system function GetSprite. A call to GetSprite takes the form

```
spritenum = GetSprite(ssp,num);
```

where ssp is a pointer to a SimpleSprite data structure, and num is the specific sprite number that you want to use (a number from 0 to 7).

If the specified sprite has not already been reserved for use by another task, the system will return the value of spritenum the same as the num value you requested. If the sprite is already reserved, the system returns a spritenum of −1, which means you have to try again.

You use a num value of −1 to request any available sprite. If this request returns with a value of −1 for spritenum, you know you are out of sprites entirely. Here is a program fragment that just asks for any sprite:

```
struct SimpleSprite mysprite;
BYTE spritenum;

spritenum = GetSprite(&mysprite, -1);

if(spritenum == -1) printf("OUT OF SPRITES!!!");
```

The GetSprite function fills in the num data field of the SimpleSprite data structure. You receive the spritenum value simply as a check that all went OK. You can use this value later to determine that sprite's colors.

Changing a Sprite

Other elements of the SimpleSprite data structure are initially established by the ChangeSprite routine. A call to ChangeSprite takes the form

```
ChangeSprite(pointer,addrsprite,addrdata);
```

where pointer contains the address of a ViewPort structure or a value of zero. If it contains the address of a viewport, then the sprite will be positioned relative to the top-left corner of the viewport. In the simple sprite program, you will see how the viewport address is extracted from a Screen data structure. If you use Intuition to establish your Screen structure, and therefore your ViewPort data structure, your sprites will move when the screen stops moving (as you will see when you try the example).

If the pointer value is zero, then the sprites will be positioned with respect to the View data structure. The View defines the overall display area, rather than individual screens and windows. Your sprites will be positioned relative to the upper-left corner of the physical display selected when Preferences was run. When the pointer is zero, sprites can move across screen boundaries. This creates a rather odd effect: the sprite takes on the colors of the screen over which it appears.

The other two parameters to the ChangeSprite function call are addrsprite and addrdata. The addrsprite parameter is a pointer to the address of a SimpleSprite data structure. This data structure should

already have been initialized by GetSprite to contain a valid sprite number. The addrdata parameter is a pointer to the first word of the data structure that describes the physical appearance of the sprite itself, i.e., the bit pattern (16 bits wide) that is used by the system to define the shape of the sprite.

The Sprite Data

Here is the data structure pointed to by the addrdata parameter to ChangeSprite. It defines the sprite's shape. It has no specific name, but its elements relate to the SimpleSprite data structure.

```
/* spritedata.c */

UWORD sprite_data[ ] = {
                        0,0,                    /* position control */
                        0x0fc3, 0x0000,         /* image data line 1 */
                        0x3ff3, 0x0000,         /* image data line 2 */
                        0x30c3, 0x0000,         /* image data line 3 */
                        0x0000, 0x3c03,         /* image data line 4 */
                        0x0000, 0x3fc3,         /* image data line 5 */
                        0x0000, 0x03c3,         /* image data line 6 */
                        0xc033, 0xc033,         /* image data line 7 */
                        0xffc0, 0xffc0,         /* image data line 8 */
                        0x3f03, 0x3f03,         /* image data line 9 */
                        0,0                   /* end of the structure */
                        };
```

A simple sprite's data always takes the form shown above. There are two words (32 bits) of zeros at the head of the structure, followed by as many pairs of data words as there are lines (tall) in the sprite, finally followed by another two words of zeros.

The first two words are for position control. For a hardware sprite, these establish where the sprite appears on the screen (and control a few other characteristics that are not discussed here). The last two words are also position control for the hardware sprite, but they are used to establish the end of the use of the sprite for a single display screen. Simple sprites can only be used once per screen. The sprite data pattern shown here forms the characters

S!

The data that the simple sprite works on must be in chip-accessible memory. That is, it must be located in the lowest 512K of the machine. The simple sprite program in this chapter uses the above data for all the

sprites. It allocates MEMF_CHIP memory and copies the data into that memory space. Subsequently, a call to ChangeSprite shows the system where to find each unique occurrence of the data area for each sprite. A separate data area must be provided for each sprite because the system modifies the position control items in each SimpleSprite data structure, then tells the hardware sprite where to find its data.

Sprite Colors

For the lines in between the two position controls, each pair of words defines the color value to be used for the particular bit position in that line of the sprite. Take image data line 1 as an example:

```
0x0fc3, 0x0000,
```

Here, 0x0fc3 translates to this set of binary bits:

0000111111000011

0x0000 becomes:

0000000000000000

Bits in corresponding positions combine to select the color for a particular pixel of that line of the display. This is how the combinations are formed:

Value in first word:	0 0 1 1
Value in second word:	0 1 0 1
Color number selected:	t 1 2 3

The t stands for a color combination that is treated as transparent. In any region of a sprite that is transparent, you can see every other item that has a lower video priority.

Simple sprites 0 and 1 use the same system color registers. Sprites 2 and 3 are paired, sprites 4 and 5 are paired, and sprites 6 and 7 are paired. Each pair of sprites has a group of system color registers specifically assigned. These define the colors that it can display. Table 7.1 shows the grouping.

Thus, in the simple sprite system, you actually have the choice of only four completely different sets of colors for your sprite, and you must also remember that the sprite colors are grouped as shown.

Sprites	Color Number	System Color Register Number
0 and 1	1	17
	2	18
	3	19
2 and 3	1	21
	2	22
	3	23
4 and 5	1	25
	2	26
	3	27
6 and 7	1	29
	2	30
	3	31

Table 7.1: Simple sprite color registers

Intuition uses simple sprite number 0 as its cursor. Therefore, that sprite is generally not available for you to assign. When you use LoadRGB4 to establish your own custom colors for a screen, if you load color numbers 17–19, you will affect the color of the system cursor as well as your other sprites.

The simple sprite program has an interesting effect on the colors of the mouse cursor as well as on the simple sprite elements. The program defines half of the sprites as positioned relative to the viewport (that is, relative to the screen where the window is drawn), and the other half as positioned relative to the display.

If you use the cursor to move into the custom screen's title bar, hold down the left mouse button, and drag the screen down about a third of the way, you will see that the relative positions of the sprites change. And some of the sprites will cross the screen border (as does the mouse cursor) and change colors as they do so.

This color change happens because Intuition dynamically assigns colors to the system color registers so as to allow each screen to have its own unique set of up to 32 colors. Thus, the screen boundary is a position at which the colors can change.

Freeing a Sprite

If you use GetSprite to obtain the use of a sprite, you must also free the sprite when you are finished with it. The system does not keep track of which task allocated a sprite and whether that task is still running. If you do not free a sprite, then no other application will be able to use it until the next system reset.

A call to FreeSprite takes the form

```
FreeSprite(num)
```

where num is the number of a sprite that you allocated with GetSprite.

The Simple Sprite Program

Listing 7.1 is the program that demonstrates the features of the simple sprite system. A 32-color custom screen is established, with a window on that screen to provide a close gadget to stop the program and a place to receive timing messages (INTUITICKS) once each one tenth of a second.

Note that simple sprites can move a lot faster than this. You may want to modify the program to wait for Intuition events and timing events (as mentioned in Chapter 6), just to make it run faster.

There are a few things you may notice as you run the program. As it begins, the leftmost sprite is represented by hardware sprite number 1. The allocations proceed across the screen to the right as hardware sprite numbers 2, 3, 4, 5, 6, and 7. (You may want to change the sprite image data to make the sprites actually contain the hardware sprite number that was allocated for that image.)

The video priorities of the hardware sprites are set such that the lowest numbered sprite has the highest video priority. In other words, sprite 0, the mouse cursor, will always appear in front of any other sprite. Sprite 1 appears in front of any sprite except 0, and so on. The system is also set so that all sprites appear in front of anything in the playfield (i.e., screen) area. If you want to modify system priorities so that some sprites occasionally disappear behind certain background elements, it is possible to do so. See the *Amiga Hardware Manual* for details.

Drag down the custom screen and watch what happens to the sprite positions and colors. That experiment will help you to understand how to use MoveSprite and ChangeSprite to position your simple sprites.

Notice one difference in the listing of the NewScreen structure from what you have seen previously in this book. That is, the Modes variable has been set to a value of SPRITES. This tells Intuition and its supporting routines that sprites are to be present and that instructions to support sprite movements should be produced when the screen is constructed.

```
/* simplesprite.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"

SHORT sprgot[7];                /* which sprite gotten each time. */
UWORD *sprdata[7];              /* seven pointers to sprite data */
SHORT xmove[7], ymove[7];       /* their directions of movement */
struct SimpleSprite sprite[7];  /* seven simple sprites */
struct SimpleSprite *spr;       /* pointer to a sprite */
short maxgot;                   /* max # of sprites we grabbed */

struct Window *w;       /* pointer to a Window   */
struct RastPort *rp;    /* pointer to a RastPort */
struct Screen *s;       /* pointer to a Screen */
struct ViewPort *vp;    /* pointer to a ViewPort */

struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* This is "prototype" data.  Each sprite will start out with
 * this data, but the position control values will be changed
 * by the simple sprite system for each one of the sprites
 * independently.  Thus it is not possible to point more
 * than one simple sprite to the same set of instance data.
 * The other reason this is only prototype data is that
 * the sprite data MUST be in CHIP-accessible RAM.  If your
 * Amiga has expanded RAM, the program will likely load
 * into expanded memory where the chips cannot get at
 * the data that is part of the program.  Thus the
 * specific allocation of MEMF_CHIP done below. */

/* 22 words of sprite data = 44 bytes of sprite data */

UWORD sprite_data[ ] = {
                        0,0,            /* position control */
                        0x0fc3, 0x0000, /* image data line 1*/
                        0x3ff3, 0x0000, /* image data line 2*/
                        0x30c3, 0x0000, /* image data line 3*/
                        0x0000, 0x3c03, /* image data line 4*/
                        0x0000, 0x3fc3, /* image data line 5*/
                        0x0000, 0x03c3, /* image data line 6*/
                        0xc033, 0xc033, /* image data line 7*/
                        0xffc0, 0xffc0, /* image data line 8*/
                        0x3f03, 0x3f03, /* image data line 9*/
                        0,0 /* end of the structure */

                        /* Structure end is "position control" for
                         * the next reuse of the hardware sprite.
                         * Simple sprite system supports only one
                         * use of a hardware sprite per video frame.
                         * Any combination of binary bits from word
                         * 1 and word 2 per line establishes the
                         * color for a pixel on that line.
                         * Any nonzero pixels in lines 1-3 are color
                         * "1" of the sprite, lines 4-6 are color "2",
                         * lines 7-9 are color "3"   */
                        };
```

Listing 7.1: The simple sprite program

```
/*
   FOLLOWING IS FOR INFORMATION ONLY.... the simple-sprite
   system directly sets these bits;  the user has no need
   to fiddle with any of them.  Use the functions
   ChangeSprite and MoveSprite to have an effect on the sprite.

   position control:

        first UWORD:
                bits 15-8, start vertical value, lowest 8 bits
                                of this value contained here.
                bits  7-0, start horizontal value, highest 8 bits
                                of this value contained here.

        second UWORD:
                bits 15-8, end (stopping) vertical value, lowest
                                8 bits of this value contained here.
                bit 7 = Attach-bit (used for attaching sprites to
                                get additional colors (15 instead
                                of 3, supported by the hardware but
                                NOT supported by the simple sprite
                                system).
                bits 6-4 (unused)
                bit 2   start vertical value; bit 8 of that value.
                bit 2   end vertical value; bit 8 of that value.
                bit 2   start horizontal value; bit 0 of that value.
*/

movesprites()
{
        short i;
        short newx, newy;

        /* MoveSprite first parameter equals zero; sprite is positioned
         * relative to the VIEW rather than the viewport.  This means
         * that if the screen is dragged down while the sprite is
         * moving, the sprite should stay in the same place as it
         * was before the drag.  If the zero is replaced by "vp",
         * then the sprite stays with the screen.
         */

        spr = &sprite[0];
for (i=0; i<maxgot; i++)
{
        newx = xmove[i]+spr->x;
        newy = ymove[i]+spr->y;

        /* This may look strange, but it shows the options of
         * moving a sprite with respect to a viewport or with
         * respect to the overall display.  Half of the sprites
         * stay with the display if you drag the screen down;
         * the other half move with respect to the overall
         * display.  When the sprites enter a different screen,
         * they take on whatever colors that screen has.
         */
        if( (i % 2) == 0 )
        {
                /* even-numbered sprites move with display */
                MoveSprite(0,spr,newx,newy);
        }
        else
        {
                /* odd-numbered sprites move with Screen */
                MoveSprite(vp,spr,newx,newy);
```

Listing 7.1: The simple sprite program (continued)

```
                }

                if(spr->x >= 320 || spr->x <= 0) xmove[i]=-xmove[i];
                if(spr->y >= 190 || spr->y <= 0) ymove[i]=-ymove[i];

                spr++;  /* adjust pointer to do next sprite */
        }
}


#define AZCOLOR 1
#define WHITECOLOR 2

/* #include "mydefines.h"*/
/* gets the window flags assignments */
/* mydefines.h */

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP|WDR)


/* #include "myscreen1.h"  */
/* myscreen1.h */

/* myfont1 specifies characteristics of the default font;
 * this case selects the 80-column font that displays as
 * 40 columns in low-resolution mode.
 */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,           /* LeftEdge, TopEdge ... where to put screen */
        320, 200,       /* Width, Height ... size of the screen */
        5,              /* 5 planes Depth, means 2 to the 5th or
                        * 32 different colors to choose from once
                        * the screen is opened.
                        */
        1, 0,           /* DetailPen, BlockPen */
        SPRITES,        /* ViewModes ... value of 0 = low resolution */
        CUSTOMSCREEN,   /* Type of screen */
        &myfont1,       /* Font to be used as default for this screen */
        "32 Color Test", /* DefaultTitle for its title bar */
        NULL,           /* screens user-gadgets, always NULL, ignored */
        NULL };
                        /* address of custom bitmap for screen,
                         * not used in this example
                         */


/*      #include "window1.h"  */
/* window1.h */

struct NewWindow myWindow = {
        0,              /* LeftEdge for window measured in pixels,
                           at the current horizontal resolution,
                           from the leftmost edge of the Screen */
        15,             /* TopEdge for window measured in lines
                           from the top of the current Screen.   */
        320, 150,       /* Width, Height of this window */
```

Listing 7.1: The simple sprite program (continued)

```
        0,              /* DetailPen - what pen number is to be
                           used to draw the borders of the window */
        1,              /* BlockPen - what pen number is to be
                           used to draw system-generated window
                           gadgets */
                        /* (for DetailPen and BlockPen, the value
                            of -1 says "use the default value")  */
        CLOSEWINDOW | INTUITICKS,
                        /* IDCMP Flags */
        SIMPLE_REFRESH | NORMALFLAGS | GIMMEZEROZERO | ACTIVATE,
                        /* Window Flags */
        NULL,           /* FirstGadget */
        NULL,           /* CheckMark   */
        "Click Close Gadget To Stop",   /* Window title */
        NULL,           /* Pointer to Screen if not workbench */
        NULL,           /* Pointer to BitMap if a SUPERBITMAP window */
        10, 10,         /* minimum width, minimum height */
        320, 200,       /* maximum width, maximum height */
        CUSTOMSCREEN
        };
#include "graphics/gfxmacros.h"

/* #include "eventl.c"  */
/* gets the event handler */
/* eventl.c */

HandleEvent(code)
        LONG code;      /* provided by main */
{
        switch(code)
        {
        case CLOSEWINDOW:
                return(0);
                break;
        case INTUITICKS:
                movesprites();  /* 10 moves per second; test */
                default:
                        break;
        }
return(1);
}

UWORD mycolortable[] = {

        0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
        0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
        0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

        0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
        0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
        0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df
        };

        /* black, red, white, fire-engine red, orange, yellow,
        lime green, green, aqua, dark blue, purple,
        violet, tan, brown, gray, skyblue, (everything again) */

main()
{
        struct IntuiMessage *msg;
        LONG result;
        short k, j;
        UWORD *src, *dest;      /* for copying sprite data to RAM */
```

Listing 7.1: The simple sprite program (continued)

```
GfxBase = OpenLibrary("graphics.library",0);

IntuitionBase = OpenLibrary("intuition.library",0);
/* (error checking left out for brevity here) */

s = OpenScreen(&myscreen1);  /* try to open it */
if(s == 0)
{
        printf("Can't open myscreen1\n");
        exit(10);
}
myWindow.Screen = s;    /* say where screen is located */

w = OpenWindow(&myWindow);
if(w == 0)
{
        printf("Window didn't open!\n");
        CloseScreen(s);
        exit(20);
}
vp = &(s->ViewPort);
/* set the colors for this viewport */

LoadRGB4(vp, &mycolortable[0], 32);
rp = w->RPort;
        /* Now wait for a message to arrive from Intuition
         * (task goes to sleep while waiting for the message)
         */

/* ************************************* */
/* SIMPLE SPRITE DEMO SECTION            */

maxgot = 0;             /* how many sprites did we get? */

spr = &sprite[0];       /* address of the first simple sprite */

for(k=0; k<7; k++)
{
        xmove[k]=1;
        ymove[k]=1;

        /* get the next available sprite */
        sprgot[k] = GetSprite(spr,-1);

        if(sprgot[k] == -1) break;
        maxgot++;

        /* initialize position and size info */
        sprite[k].x = 0;
        sprite[k].y = 0;

        /* tell system what data looks like by specifying height */

        sprite[k].height = 9;

        /* now allocate CHIP memory to hold the actual sprite data */

        sprdata[k] = (UWORD *)AllocMem(44, MEMF_CHIP);

        if(sprdata[k] == NULL)
        {
                maxgot--;
                FreeSprite(sprgot[maxgot]);
```

Listing 7.1: The simple sprite program (continued)

```
                break;  /* if not enought memory, stop allocating
                         * any more sprites but try to carry on
                         * anyway.
                         */
        }

        /* now copy the prototype sprite data into the CHIP RAM. */

        src = sprite_data;  dest = sprdata[k]; /* source, destination */

        for( j=0; j<22; j++)
        {
                *dest++ = *src++;
        }
        /* tell system sprite manager where to find sprites data */

        ChangeSprite(vp,spr,sprdata[k]);

        /* choose starting point based on which sprite this is */

        MoveSprite(0,spr,10 + 20*sprgot[k],30);

        spr++;  /* do the next sprite now */
}

while(1)        /* "forever" */
{
        /* wait for a message to arrive */
        WaitPort( w->UserPort );

        /* retrieve the message from the port */
        msg = (struct IntuiMessage *)GetMsg(w->UserPort);
handleit:
        result = -1;
        if(msg != 0)
        {
                /* handle the event; see if CLOSEWINDOW */
                result = HandleEvent(msg->Class);

                /* Let Intuition reuse the msg */
                ReplyMsg(msg);
        }
        if(result == 0)
        {
                break;  /* got a CLOSEWINDOW */
        }
        /* Empty the port before going back to wait again */

        msg = (struct IntuiMessage *)GetMsg(w->UserPort);

        if(msg != 0)
        {
                goto handleit;  /* 0 when no more messages */
        }
}
/* Done! Now cleanup. */

/* free however many sprites we actually managed to get */
for(k=0; k<maxgot; k++)
{
        FreeSprite(sprgot[k]);  /* free the sprites so others can
                                 * use them also
                                 */
```

Listing 7.1: The simple sprite program (continued)

```
            FreeMem(sprdata[k],44); /* and free the memory they used */
        }

        CloseWindow(w);
        CloseScreen(s);
}
```

Listing 7.1: The simple sprite program (continued)

VIRTUAL SPRITES

The animation system is built as a set of levels of software. The simple sprite system is closest to the hardware system, and has the same limitations as the hardware itself. Without using special tricks, there are only eight hardware sprites available, so you only get eight per display.

Virtual sprite handling by the gel system is slightly more complicated. It is as close to the hardware as the simple sprite system at its level of operation. However, it interfaces to the hardware in a somewhat different way. A virtual sprite is defined, as the name "virtual" implies, as software elements. Virtual sprites become "real" sprites when the system software assigns each to be displayed by a hardware sprite.

Instead of having only eight possible sprites on the screen at a time, you might be able to have 16, 24, 32, or even over 100, depending on how large each sprite is and where it is located. Admittedly, that might be pushing things a little, but you can experiment to see the actual limits for your applications.

The gel system knows that there are eight hardware sprites available, and it also knows how to arrange to reuse each sprite many times in the span of a single display.

A virtual sprite (VSprite) data structure includes definitions for the position of a sprite, its size, its bit pattern, and the colors that it should display. Every time a particular sprite has completely displayed its assigned image, it becomes free to display a new image in a new position on the screen a couple of lines further down than the end of the previous image and anywhere else horizontally.

The system not only tells the hardware sprite to redisplay itself somewhere else with a different bit pattern. It also gives the hardware sprite a new set of colors from a virtual sprite definition to use for the next image.

Advantages to Using Virtual Sprites

The main advantage to using virtual sprites is that you do not have to worry about exactly how the system allocates sprites. That happens automatically. You just define how the sprites are to appear and the system does the rest. Another advantage is that with the system's reuse of the sprites, you have more than eight sprites available.

Disadvantages to Using Virtual Sprites

If you ask the system to display more than eight virtual sprites on a single horizontal line, some of the sprites may disappear. This is a hardware limitation. Once all eight hardware sprites are busy, you can't get any more displays out of them than one set of 16 bits per horizontal line.

Another disadvantage is that you don't really have eight hardware sprites to work with after all. In particular, as noted, hardware sprite 0 is reserved for use by Intuition as the mouse pointer. Since hardware sprite 0 and hardware sprite 1 both share the same set of color registers, it is inappropriate for hardware sprite 1 to be used as a virtual sprite because this would result in the colors of the mouse cursor flashing as sprite 1 was assigned to display various virtual sprites on the screen.

A third disadvantage comes in the area of video priorities. Recall that each simple sprite is directly related to a hardware sprite, with the lower numbered sprites having video priority over the higher numbered sprites. When you use virtual sprites, the system arbitrarily assigns hardware sprites to display the virtual sprites. As your virtual sprites move across one another, depending on their relative positions on the screen, it is likely that the sprites will swap video priorities. That is, a sprite that just one moment ago was behind another sprite will now be in front of that sprite. This is a result of the dynamic reassignment of sprites as the system runs.

There is one more item to contend with when you are using virtual sprites: the color choices available to your program for drawing items in the background area. You may wish to limit your playfield area (that is, your screen) to a maximum of 16 colors. Alternatively, you can specify a maximum of 32 colors, but (assuming that you do not use hardware sprites 0 and 1 because of the mouse cursor color flashing mentioned above) restrict your screen color usage to colors 0–20, 24, and 28. In other words, you will not use colors 21–23, 25–27, or 29–31.

As the gel system assigns virtual sprites to hardware sprites, it also assigns a particular virtual sprite's colors to a particular hardware

sprite's color registers. Thus, anything you have drawn in that color will continuously change to match the color currently assigned to that sprite. The makevsprite program shown later outputs text material to demonstrate color register reassignment.

Initializing the Gel System

To use virtual sprites or any other part of the gel system, you must first initialize the system to expect gels. A routine called ReadyGels is used for this purpose. It is shown in Listing 7.2. You must also declare a data structure called GelsInfo. This contains the data that the gel system will need to keep track of your virtual sprites as well as your Bobs. Once you have opened a screen, ReadyGels can be run.

SpriteHead and SpriteTail

In Listing 7.2, you will see the terms SpriteHead and SpriteTail. The system builds a list of all of the gels in the system and their display sequence. Gels are sorted from top to bottom and from left to right. SpriteHead and SpriteTail are the two ends of this gels list.

Reserved Sprites

You can tell the system which particular hardware sprites to use as virtual sprites. Setting particular binary bits to a value of 1 tells the system that it is OK to dynamically reassign that sprite and its color registers as a virtual sprite. The bits of the reserved sprite parameter (sprRsrvd) are numbered to match the sprites themselves. In other words, sprite 7 is represented by bit 7, sprite 6 by bit 6, and so on.

The form of ReadyGels in the listing uses a bit pattern of 0xFC, which has a binary value of 11111100. This means "do not use sprites 0 and 1." What you might consider doing is changing the sprRsrvd value to match that of the simple sprite system, located at GfxBase->SpriteReserved. You see, every time the simple sprite system allocates a sprite for a task's use, it sets a bit in SpriteReserved in GfxBase. If a bit is a one in that variable, the corresponding bit should be a zero here.

Here is a sequence that will accomplish the desired result:

```
struct GfxBase *GfxBase;
/* used as an alternative to the declaration */
/* LONG GfxBase; that has been used elsewhere in this book. */

/* ... then in ReadyGels, in place of g->sprRsrvd = 0xFC */

g->sprRsrvd = 0xFC & (!(GfxBase->SpriteReserved));

/* set any bit to zero for which there is a 1-bit in the */
/* simple sprite system. */
```

```
/* readygels.c */

struct VSprite *SpriteHead = NULL;
struct VSprite *SpriteTail = NULL;

ReadyGels(g, r)
struct RastPort *r;
struct GelsInfo *g;
{
   /* Allocate head and tail of list. */
   if ((SpriteHead = (struct VSprite *)AllocMem(sizeof
         (struct VSprite), MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      return(-1);
   }

   if ((SpriteTail = (struct VSprite *)AllocMem(sizeof
         (struct VSprite), MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      FreeMem(SpriteHead, sizeof(struct VSprite));
      return(-2);
   }
   g->sprRsrvd = 0xFC;  /* do not use sprites 0 or 1. */

   if ((g->nextLine = (WORD *)AllocMem(sizeof(WORD) * 8,
         MEMF_PUBLIC | MEMF_CLEAR)) == NULL)
   {
      FreeMem(SpriteHead, sizeof(struct VSprite));
      FreeMem(SpriteTail, sizeof(struct VSprite));
      return(-3);
   }

  if ((g->lastColor = (WORD **)AllocMem(sizeof(LONG) * 8,
         MEMF_PUBLIC | MEMF_CLEAR)) == NULL)
   {
      FreeMem(g->nextLine,8 * sizeof(WORD));
      FreeMem(SpriteHead, sizeof(struct VSprite));
      FreeMem(SpriteTail, sizeof(struct VSprite));
      return(-4);
   }

  /* Next we prepare a table of pointers to the routines that should
   * be performed when DoCollision senses a collision.  This
   * declaration may not be necessary for a basic vsprite with
   * no collision detection implemented, but then it makes for
   * a complete example. */

   if ((g->collHandler = (struct collTable *)AllocMem(sizeof(struct
         collTable), MEMF_PUBLIC | MEMF_CLEAR)) == NULL)
   {
      FreeMem(g->lastColor, 8 * sizeof(LONG));
      FreeMem(g->nextLine,8 * sizeof(WORD));
      FreeMem(SpriteHead, sizeof(struct VSprite));
      FreeMem(SpriteTail, sizeof(struct VSprite));
      return(-5);
   }

  /* When any part of the object touches or passes across
   * this boundary, it will cause the boundary collision
   * routine to be called.  This is at smash[0] in the
   * collision handler table and is called only if
   * DoCollision is called. */
```

Listing 7.2: The readygels routine

```
    g->leftmost = 0;
    g->rightmost = r->BitMap->BytesPerRow * 8 - 1;
    g->topmost = 0;
    g->bottommost = r->BitMap->Rows - 1;

    r->GelsInfo = g;  /* Link together the two structures */

    InitGels(SpriteHead, SpriteTail, g );

   /* Pointers initialized to the dummy sprites which will be
    * used by the system to keep track of the animation system. */

    SetCollision(0, border_dummy, g);
    WaitTOF();
    return(0);    /* a return value of 0 says all OK, any
                   * negative value tells you when it failed.
                   * (see the listing as to what the routine
                   * was trying to do... all failures are
                   * due to out of memory conditions). */

}

void border_dummy()     /* a dummy collision routine */
{
    return;
}
```

Listing 7.2: The readygels routine (continued)

This technique is necessary because the simple sprite system was developed independently from the gel system. As of system software release version 1.2, they remain independent of one another.

Next Lines and Last Colors

The system needs certain variables to enable it to decide which hardware sprite to use for the display of the next virtual sprite. These status variables are parameters in the GelsInfo structure and include an array of "next lines" and an array of "last colors."

The nextline array is used to hold system information about the line number on the screen at which each hardware sprite will become available to be given a new virtual sprite to display.

In the lastcolor pointer array, the system stores a pointer to the color definitions most recently used. The virtual sprite colors are written into the hardware sprite register set for the hardware sprite to which that virtual sprite is assigned. This array contains one pointer to the last set of three colors (from the VSprite structure sprColors pointer) for each hardware sprite.

As the system is scanning to determine which hardware sprite should next be used to represent a virtual sprite, it checks the contents of the lastcolor array. If a hardware sprite is available and has been assigned

this set of colors, no color assignment is needed, and therefore no color change instructions will be generated for the Copper list—a coprocessor instruction list. (See the *Amiga Hardware Manual* for information about the Copper coprocessor.)

If all of your virtual sprites use a different set of colors (i.e., if the pointers to sprColors are different for each of them), then you are limited to four virtual sprites per horizontal line. If, on the other hand, you define eight virtual sprites, with 0 and 1 having the same colors, 2 and 3 the same as each other, 4 and 5 the same as each other, and 6 and 7 the same as each other, then you will be able to have all eight virtual sprites on the same horizontal line.

Since the system hardware shares the color registers between pairs of hardware sprites, it thus has enough resources to assign eight virtual sprites to hardware sprites because there are four color sets for eight virtual sprites, exactly matching the maximum hardware capabilities. (Note that lastcolor is not used for Bobs, just for sprites.)

The MakeVSprite Routine

Listing 7.3 contains the MakeVSprite routine, which helps define a virtual sprite. Although MakeVSprite creates a virtual sprite, it does not add the sprite to the system list for possible display. That must be done by the calling program. If MakeVSprite returns a zero value, there was not enough memory to create the virtual sprite. A nonzero value is the address of a virtual sprite that can now be added to the system with AddSprite.

The VSprite Structure

The MakeVSprite routine uses the parameters to the VSprite structure. This structure is used to define both virtual sprites and Bobs. A VSprite data structure is dynamically allocated, and the variables that you provide are installed in the structure. The structure also allocates and initializes some collision masks and collision variables, assuming there is a default boundary collision routine present (border_dummy is provided in the listing). Following are descriptions of the parameters to the VSprite structure.

Height

The lineheight parameter specifies how tall in lines this sprite is to be. You can specify any number of lines. Your bit image of the sprite must have this many lines. (For a Bob, the height can also be any number of lines. The taller the Bob, the longer the system takes to draw it.)

```
/* makevsprite.c */

struct VSprite *MakeVSprite(lineheight, image, colorset, x, y,
                            wordwidth, imagedepth, flags)
SHORT lineheight;       /* How tall is this vsprite? */
WORD *image;            /* Where is the vsprite image data; should be
                           twice as many words as the value of lineheight */
WORD *colorset;         /* Where is the set of three words that describes
                           the colors this vsprite can take on? */
SHORT x, y;             /* What is its initial on-screen position? */
SHORT wordwidth, imagedepth, flags;
{
   struct VSprite *v;   /* Make a pointer to the VSprite structure that
                           this routine dynamically allocates */

   if ((v = (struct VSprite *)AllocMem(sizeof(struct VSprite),
        MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      return(0);
   }

   v->Flags = flags;    /* Is this a vsprite, not a Bob? */

   v->Y = y;            /* Establish initial position relative to */
   v->X = x;            /* the Display coordinates. */

   v->Height = lineheight;      /* The Caller says how high it is. */
   v->Width = wordwidth;        /* A vsprite is always 1 word (16 bits) wide. */

  /* There are two kinds of depth... the depth of the image itself, and the
   * depth of the playfield into which it will be drawn. The image depth
   * says how much data space will be needed to store an image if it's
   * dynamically allocated. The playfield depth establishes how much space
   * will be needed to save and restore the background when a Bob is drawn.
   * A vsprite is always 2 planes deep, but if it's being used to make a
   * Bob, it may be deeper... */

   v->Depth = imagedepth;

  /* Assume that the caller at least has a default boundary collision
   * routine.... bit 1 of this mask is reserved for boundary collision
   * detect during DoCollision(). The only collisions reported will be
   * with the borders. The caller can change all this later. */

   v->MeMask = 1;
   v->HitMask = 1;

   v->ImageData = image;        /* Caller says where to find the image. */

  /* Show system where to find a mask which is a squished down version
   * of the vsprite (allows for fast horizontal border collision detection). */

   if ((v->BorderLine = (WORD *)AllocMem((sizeof(WORD)*wordwidth),
        MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
        FreeMem(v, sizeof(struct VSprite));
        return(0);
   }

  /* Show system where to find the mask that contains a 1-bit for any
   * position in the object in any plane where there is a 1-bit (all planes
   * OR'ed together). */
```

Listing 7.3: The makevsprite routine

```
    if ((v->CollMask = (WORD *)AllocMem(sizeof(WORD)*lineheight*wordwidth,
          MEMF_CHIP | MEMF_CLEAR)) == 0)
    {
         FreeMem(v, sizeof(struct VSprite));
         FreeMem(v->BorderLine, wordwidth * sizeof(WORD));
         return(0);
    }

   /* This isn't used for a Bob, just a VSprite. It's where the
    * Caller says where to find the VSprites colors. */

    v->SprColors = colorset;

    /* These aren't used for a VSprite, and MakeBob'll do set up for Bob. */
    v->PlanePick  = 0x00;
    v->PlaneOnOff = 0x00;

    InitMasks(v);        /* Create the collMask and borderLine */
    return(v);
}
/* end of listing 7.3 - Makevsprite.c */
```

Listing 7.3: The makevsprite routine (continued)

Image

The image parameter is a pointer to the array of data words that defines the physical appearance of your sprite. The sprite image that was shown in Listing 7.1 is a typical image. The address at which the image starts is the location of line 1 of the image data.

Unlike simple sprites, virtual sprites do not need an individual occurrence of the sprite image data for each image of the sprite that is produced. However, as in the simple sprite system, this sprite image must be located in chip-accessible RAM.

Colors

The colorset parameter is a pointer to a set of three words (48 bits), which define the colors that are to be shown for this graphics object. These words contain the actual color data for color numbers 1, 2, and 3 of a virtual sprite.

The system keeps track, in the lastcolors array, of the pointer value used to assign colors for a particular sprite. If you have several virtual sprites with identical colors, this pointer value should be the same for all of them. When the system is trying to find a hardware sprite to assign for a particular virtual sprite, it knows that sprite pairs share the same color registers. If you load the same colors for each of the sprites of a sprite pair, it is much easier for the system to find an available hardware sprite to use. However, if all of the color pointer values are

different, you increase the chances of the system running out of hardware sprites and thus deciding not to display some of your virtual sprites in this particular display frame. (Decisions are made once each 1/60th of a second about what can be displayed.)

Position

The x and y parameters determine the on-screen position of the graphics object. Unlike simple sprites, which can be positioned relative to the display, virtual sprites are restricted to positions relative to the screen in which they are drawn. As with simple sprites, however, they do not interfere with the playfield area.

Width

The wordwidth parameter specifies the width of the graphics object. For a virtual sprite, this value is always 1. (A Bob can be any width; the wider the Bob, the longer the system might take to draw it.)

Depth

The imagedepth parameter defines how many planes of data are contained in the image area. For a virtual sprite, this value is always 2. (For a Bob, the depth can be any value less than or equal to the depth of the screen into which it will be drawn.)

Flags

The flags identify the type of graphics object that is represented by the VSprite data structure. For a virtual sprite, set this parameter to VSPRITE. (Set it to 0 for a Bob.)

The Virtual Sprite Program

Listing 7.4 is a program that uses the routines we've created to duplicate some of the features of the simple sprite program shown earlier. The main difference between the two programs is that a lot more virtual sprites are generated here, and rather than relying on INTUITICKS, this program waits for the top of the next frame before generating the next sprite move. The sprites in this example, therefore, move about six times faster than in the previous example.

The initial placement of the sprites is random compared to the simple sprite example, and enough sprites have been generated to ensure that sometimes one or two disappear for a frame or two. Watch carefully and you will see it.

Also notice the occasional change of priorities as the sprites pass beyond each other, and the change in the colors of the numerals 21–23,

```
/* vsprite.c */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"
#include "graphics/gels.h"

/* ask system to create and manage MAXSP vsprites */
#define MAXSP 28

/* define possible speeds for vsprites in counts per vblank */

SHORT speed[] = { 1, 2, -1, -2 };

SHORT xmove[MAXSP], ymove[MAXSP];
                              /* sprite directions of movement */
struct VSprite *vsprite[MAXSP]; /* MAXSP simple sprites */
struct VSprite *vspr;         /* pointer to a sprite */
short maxgot;                 /* max # of sprites we created */

struct GelsInfo mygelsinfo;   /* the window's RastPort needs one
                               * of these in order to do VSprites */

struct Window *w;       /* pointer to a Window   */
struct RastPort *rp;    /* pointer to a RastPort */
struct Screen *s;       /* pointer to a Screen */
struct ViewPort *vp;    /* pointer to a ViewPort */

struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* 18 words of sprite data = 9 lines of sprite data */

UWORD sprite_data[ ] = {
                        0x0fc3, 0x0000, /* image data line 1*/
                        0x3ff3, 0x0000, /* image data line 2*/
                        0x30c3, 0x0000, /* image data line 3*/
                        0x0000, 0x3c03, /* image data line 4*/
                        0x0000, 0x3fc3, /* image data line 5*/
                        0x0000, 0x03c3, /* image data line 6*/
                        0xc033, 0xc033, /* image data line 7*/
                        0xffc0, 0xffc0, /* image data line 8*/
                        0x3f03, 0x3f03, /* image data line 9*/
                        };
UWORD *sprdata;

movesprites()
{
        short i;
        for (i=0; i<maxgot; i++)
        {
                vspr = vsprite[i];

                vspr->X = xmove[i]+vspr->X;
                vspr->Y = ymove[i]+vspr->Y;

                /* move the sprites ... here. */

                if(vspr->X >= 300 || vspr->X <= 0) xmove[i]=-xmove[i];
                if(vspr->Y >= 190 || vspr->Y <= 0) ymove[i]=-ymove[i];
```

Listing 7.4: The virtual sprite program

```
        }
        SortGList(rp);      /* get the list in order */
        DrawGList(rp, vp); /* create the sprite instructions */

        MakeScreen(s);      /* ask Intuition to pull it all together */
        RethinkDisplay();  /* and to show us what we have now. */
}


#define AZCOLOR 1
#define WHITECOLOR 2

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP)
/* Did not use windowdrag because don't want screen to be moved. */

/* Allow window sizing so user can decrease size of window, then
 * increase it again, thus erasing the background text. User can
 * easily see that some vsprites wink out and into existence when
 * too many sprites are on a single horizontal plane and the
 * vsprite system runs out of sprites to assign.
 */

/* myfont1 specifies characteristics of the default font;
 * this case selects the 80-column font that displays as
 * 40 columns in low-resolution mode.
 */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,           /* LeftEdge, TopEdge ... where to put screen */
        320, 200,       /* Width, Height ... size of the screen */
        5,              /* 5 planes Depth, means 2 to the 5th or
                        * 32 different colors to choose from once
                        * the screen is opened.
                        */
        1, 0,           /* DetailPen, BlockPen */
        SPRITES,        /* ViewModes ... value of 0 = low resolution */
        CUSTOMSCREEN,   /* Type of screen */
        &myfont1,       /* Font to be used as default for this screen */
        "32 Color Test", /* DefaultTitle for its title bar */
        NULL,           /* screens user-gadgets, always NULL, ignored */
        NULL };
                        /* address of custom bitmap for screen,
                         * not used in this example
                         */


struct NewWindow myWindow = {
        0,              /* LeftEdge for window measured in pixels,
                           at the current horizontal resolution,
                           from the leftmost edge of the Screen */
        0,              /* TopEdge for window is measured in lines
                           from the top of the current Screen.   */
        320, 185,       /* Width, Height of this window */
        0,              /* DetailPen - what pen number is to be
                           used to draw the borders of the window */
        1,              /* BlockPen - what pen number is to be
                           used to draw system-generated window
                           gadgets */
```

Listing 7.4: The virtual sprite program (continued)

```
                        /* (for DetailPen and BlockPen, the value
                           of -1 says "use the default value")  */
        CLOSEWINDOW,    /* simplesprite program used INTUITICKS also */
                        /* IDCMP Flags */
        NORMALFLAGS | GIMMEZEROZERO | ACTIVATE,
                        /* Window Flags */
        NULL,           /* FirstGadget */
        NULL,           /* CheckMark   */
        "Click Close Gadget To Stop",   /* Window title */
        NULL,           /* Pointer to Screen if not Workbench */
        NULL,           /* Pointer to BitMap if a SUPERBITMAP window */
        320, 10,        /* minimum width, minimum height */
        320, 200,       /* maximum width, maximum height */
        CUSTOMSCREEN
        };

#include "graphics/gfxmacros.h"

/* #include "event1.c"  */
/* gets the event handler */
/* event1.c */

HandleEvent(code)
        LONG code;      /* provided by main */
{
        switch(code)
        {
        case CLOSEWINDOW:
                return(0);
                break;
        case INTUITICKS:
                movesprites();  /* 10 moves per second; test */
                default:
                        break;
        }
return(1);
}

UWORD mycolortable[] = {

        0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
        0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
        0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

        0x0000, 0x0e30, 0x0fff, 0x0b40,
        0x0fb0, 0x0bf0, 0x05d0, 0x0ed0,
        0x07df, 0x069f, 0x0c0e, 0x0f2e,
        0x0feb, 0x0c98, 0x0bbb, 0x07df
        };

        /* black, red, white, fire-engine red, orange, yellow,
        lime green, green, aqua, dark blue, purple,
        violet, tan, brown, gray, skyblue, (everything again) */

UWORD colorset0[ ] = { 0x0e30, 0xffff, 0x0b40 }; /* same as colors 17-19 */
UWORD colorset1[ ] = { 0x0bf0, 0x05d0, 0x0ed0 }; /* 21-23 */
UWORD colorset2[ ] = { 0x069f, 0x0c0e, 0x0f2e }; /* 25-27 */
UWORD colorset3[ ] = { 0x0c98, 0x0bbb, 0x07df }; /* 29-31 */
UWORD *colorset[ ] = {
                        colorset0, colorset1,
                        colorset2, colorset3 };
int choice;
```

Listing 7.4: The virtual sprite program (continued)

```
char *numbers[] = { "17","18","19",
                    "20","21","22","23",
                    "24","25","26","27",
                    "28","29","30","31" };

#include "ram:purgegels.c"
#include "ram:readygels.c"
#include "ram:makevsprite.c"

main()
{
        struct IntuiMessage *msg;
        LONG result;
        SHORT k, j, x, y, error;
        UWORD *src, *dest;      /* for copying sprite data to RAM */

        GfxBase = OpenLibrary("graphics.library",0);

        IntuitionBase = OpenLibrary("intuition.library",0);
        /* (error checking left out for brevity here) */
        s = OpenScreen(&myscreen1);  /* try to open it */
        if(s == 0)
        {
                printf("Can't open myscreen1\n");
                exit(10);
        }
        myWindow.Screen = s;    /* say where screen is located */

        ShowTitle(s, FALSE);    /* Dont let screen be dragged down...*/

        w = OpenWindow(&myWindow);
        if(w == 0)
        {
                printf("Window didn't open!\n");
                CloseScreen(s);
                exit(20);
        }
        vp = &(s->ViewPort);
        /* set the colors for this viewport */

        LoadRGB4(vp, &mycolortable[0], 32);
        rp = w->RPort;
                /* Now wait for a message to arrive from Intuition
                 * (task goes to sleep while waiting for the message)
                 */

        /* Write Text using sprite colors so that demo can show
         * how the vsprite system stuffs the colors as it goes
         * down the screen.  Thus if using vsprites, user should
         * avoid using the color registers that the vsprites use.
         */

        /* Notice also that color numbers 17-19 are untouched.
         * That is because of the sprResrvd=0xFC in ReadyGels.
         * (Doesn't allow the virtual sprite system to access
         * either sprite 0 or 1... 0 is used by mouse cursor and
         * shares its colors with 1, so I reserved both of them.
         */

        for(j=8; j<180; j+=50)
        {
                for(k=0; k<15; k++)
                {
```

Listing 7.4: The virtual sprite program (continued)

```
                Move(rp,k*20,j);
                SetAPen(rp,k+17);       /* show 17-31 */
                /* 16, 20, 24, 28 are unaffected by vsprites
                 * because they are not used by hardware sprites */

                Text(rp,numbers[k],2);
        }
}
/* ************************************ */
/* VSPRITE DEMO SECTION                 */
/* ************************************ */

/* Allocate CHIP memory to hold the actual sprite data */
/* (necessary if ever to run on an expanded RAM Amiga) */

sprdata = (UWORD *)AllocMem(36, MEMF_CHIP);

if(sprdata == NULL)
{
        /* not enough memory for sprite */
}
/* now copy the sprite data into the CHIP RAM. */

src = sprite_data;
dest = sprdata;         /* source, destination */

for( j=0; j<18; j++)
{
        *dest++ = *src++;
}

choice = 0;
maxgot = 0;

/* Prepare the GELS system to work with VSPRITE or BOBS */

error = ReadyGels(&mygelsinfo, rp);

for(k=0; k<MAXSP; k++)  /* whatever maximum number of vsprites */
{
        xmove[k]=speed[RangeRand(4)];
        ymove[k]=speed[RangeRand(4)];

        /* establish a position for the sprite */
        x = 10 + RangeRand(280);
        y = 10 + RangeRand(170);

        /* create a vsprite */
        vsprite[k] = MakeVSprite( 9, sprdata, colorset[choice],
                                  x, y, 1, 2, VSPRITE);
        /* 9 lines high, using MEMF_CHIP image of a sprite,
         * with a particular set of colors, at an X,Y location
         * 1 word wide, 2 planes deep (all vsprites are 2 deep)
         * and it is a VSPRITE */

        if(vsprite[k] == 0)
        {
                break;  /* ran out of memory! */
        }
        AddVSprite(vsprite[k], rp);

        maxgot++;
        choice++; /* choose a different color set */
```

Listing 7.4: The virtual sprite program (continued)

```
        if(choice >= 4)
        {
                choice = 0;     /* wrap around on colorsets */
        }
}
while(1)        /* forever */
{
        WaitTOF();
        movesprites();

        result = -1;    /* now see if CLOSEWINDOW is waiting */
        msg = (struct IntuiMessage *)GetMsg(w->UserPort);
        if(msg != 0)
        {
                result = HandleEvent(msg->Class);

                /* Let Intuition reuse the msg */
                ReplyMsg(msg);
        }
        if(result == 0)
        {
                break;  /* got a CLOSEWINDOW */
        }
}

/* DONE, now cleanup */

/* Free however many vsprites we actually managed to create */

for(k=0; k<maxgot; k++)
{
        DeleteGel(vsprite[k]);
}
/* delete what ReadyGels created */
PurgeGels(&mygelsinfo);

FreeMem(sprdata, 36);   /* free what we allocated. */

CloseWindow(w);
CloseScreen(s);
```

Listing 7.4: The virtual sprite program (continued)

25–27, and 29–31 as the sprites move around the screen. These numerals are drawn in those particular color numbers in the drawing area, showing you what the gel system is doing to those color registers as the sprites move.

Note: The program in Listing 7.4 works only for system software release 1.2 and beyond. There was a problem in the gel system that has been fixed for release 1.2. Bobs function acceptably in release 1.1, but virtual sprites work incorrectly prior to version 1.2.

BLITTER OBJECTS (BOBS)

The next topic is that of the Blitter object, also called a Bob. The Bob data structure defines these objects. Like virtual sprites, Bobs are part

of the gel system. Unlike virtual sprites, which are independent of the background display, Bobs are drawn as part of the background area. You can use a Bob as a paintbrush, drawing it anywhere you wish on the background area in multiple colors. (There are as many colors in a Bob as there are in the background into which it is drawn.) Or you can move a Bob by causing it to save the background area in its enclosing rectangle before it is drawn and to restore the background area when it is moved elsewhere.

The Bob data structure contains fields that describe how a Bob is constructed and how it behaves. The structure also contains a pointer to a VSprite data structure, which contains the rest of the data definition. Having the VSprite structure as part of the overall definition of a Bob allowed the system designers to utilize a single routine to keep track of the positions of all graphics elements as part of a single animation.

The MakeBob Routine

Listing 7.5 contains the MakeBob routine. Most of the parameters for MakeBob are identical to those for MakeVSprite. Among those that are different are the image pointer and the PlanePick and PlaneOnOff parameters.

The Image Pointer

Recall that each line of a sprite is defined by two data words at consecutive memory locations, for example:

```
0x0fc3, 0x0000,                    /* image data line 1 of a sprite */
```

For a Bob, the data is different. All of the data for a particular bitplane is grouped together. When you specify the data for a Bob, it is usually most aesthetically pleasing if you group the data to resemble the physical shape of the Bob so that the bit patterns are somewhat recognizable. The difference between the color selection of sprites as compared to Bobs is that the bits that combine to specify the colors for Bobs are in corresponding bit positions in corresponding lines of the pattern definition in each of the bitplanes of the pattern. A 32-bit-wide Bob is used as the example:

```
/* plane 0 of the pattern */
0x0000, 0x1111,
0xcccc, 0xeeee,
/* plane 1 of the pattern */
0xffff, 0x0000,
0xaaaa, 0x7777
```

```
/* makebob.c */

struct Bob *MakeBob(bitwidth,lineheight,imagedepth,image,
     planePick,planeOnOff, x,y, flags)
SHORT bitwidth,lineheight,imagedepth,planePick,planeOnOff,x,y,flags;
WORD *image;
{
   struct Bob *b;
   struct VSprite *v;
   SHORT wordwidth;

   wordwidth = (bitwidth+15)/16;

  /* Create a vsprite for this Bob, it will need to be deallocated
   * later (freed) when this Bob gets deleted.
   * Note: No color set for Bobs. */

   if ((v = MakeVSprite(lineheight, image, NULL, x, y, wordwidth,
        imagedepth, flags)) == 0)
   {
      return(0);        /* not enough memory for a VSprite */
   }

   /* Caller selects the bitplanes into which the image is drawn. */
   v->PlanePick = planePick;

   /* What happens to the bitplanes into which the image is not drawn. */
   v->PlaneOnOff = planeOnOff;

   if ((b = (struct Bob *)AllocMem(sizeof(struct Bob),
        MEMF_PUBLIC | MEMF_CLEAR)) == 0)
   {
      return(0);        /* no memory for a Bob */
   }

   v->VSBob = b; /* Link together the Bob and its VSprite structures */

   b->Flags = 0; /* Not part of an animation (BOBISCOMP) and don't keep the
                    image present after bob is removed (SAVEBOB) */

  /* Tell where to save background. Must have enough space for as many
   * bitplanes deep as the display into which everything is being drawn. */

   if ((b->SaveBuffer = (WORD *)AllocMem(sizeof(SHORT) * wordwidth
        * lineheight * imagedepth, MEMF_CHIP | MEMF_CLEAR)) == 0)
   {
      FreeMem(b, sizeof(struct Bob));
      return(0);
   }

   b->ImageShadow = v->CollMask;

  /* Interbob priorities are set such that the earliest defined Bobs have
   * the lowest priority; last Bob defined is on top. */

   b->Before = NULL;    /* Let the caller worry about priority later. */
   b->After = NULL;

   b->BobVSprite = v;

  /* InitMasks does not preset the imageShadow ... caller may elect to use
   * the collMask or to create own version of a shadow, although it
   * is usually the same. */
```

Listing 7.5: The makebob routine

```
    b->BobComp = NULL;    /* this is not part of an animation */
    b->DBuffer = NULL;    /* this is not double buffered */

   /* Return a pointer to this newly created Bob for additional caller
    * interaction or for AddBob(b); */

    return(b);
}
/* end of listing 7.5, makebob.c */
```

Listing 7.5: The makebob routine (continued)

The color of the pixel in the upper-left corner of this Bob is determined by the combination of bits from the most significant bit of the words 0000 and 1111. The higher numbered bitplane's bits take on the greater significance. In other words, here are the possible color selections for bits in plane 1 and plane 0:

Bit in plane 1:	0 0 1 1
Bit in plane 0:	0 1 0 1
Color number selected:	0 1 2 3

PlanePick and PlaneOnOff

The binary bits that select color numbers are subject to modification by the values of PlanePick and PlaneOnOff. These parameters enable you to define an object that has only four possible color choices. For example, although the object discribed here has only four color choices (only two planes define the object), you can define exactly which four colors—out of a possible 32 available colors in the screen—will be used each time the object is drawn.

PlanePick is a binary bit that means, when set: "Pick these planes to be affected by the pattern, starting with the lowest bitplane. Draw the bits from the lowest numbered plane into the lowest numbered bitplane for which there is a 1-bit in PlanePick. Draw the bits from the next-to-lowest numbered plane into the next-to-lowest numbered bitplane for which there is a 1-bit in PlanePick, and so on."

For example, if PlanePick has a value of binary 0101, this is taken to mean:

Plane to be picked:	3 2 1 0
Value of PlanePick:	0 1 0 1
Use source data from image plane:	X 1 X 0

So the image data from plane 0 is written into plane 0 of the destination area, and the image data from plane 1 is written into plane 2 of the destination area.

From this scheme, there are two planes of the destination area that are unaffected by the data that is to be written: planes 1 and 3. What is to be done with the data that is already there?

This is where PlaneOnOff comes in. PlaneOnOff is also a binary value. It defines, for the planes not picked, what to do with the data. For a given bit position, if a plane is not picked and if there is a zero in that position, the plane is to be filled with zeros wherever there is a nontransparent color for the object in the source data definition. Wherever there is a 1-bit for a plane that is not picked, that plane is filled with 1-bits wherever a nontransparent color of the object exists.

The gel system generates and uses a mask for Bobs that contains 1-bits wherever there is a 1-bit in the object. By using this mask, the system can control the way it modifies the bitplanes of the display area that are unaffected by PlanePick.

If PlaneOnOff has a value of binary 0000, then the resultant color selections for the object are based on the positions of 1-bits in the pattern of the object:

0 X 0 X

from PlaneOnOff for those not picked, and

X 1 X 1

from PlanePick where pattern bits get drawn. The bit positions shown here as X indicate bit values that do not enter into the color selection process.

These are the possible color values from this combination:

0 0 0 0 = color 0 (transparent)

0 0 0 1 = color 1

0 1 0 0 = color 4

0 1 0 1 = color 5

So PlanePick and PlaneOnOff have translated the possible combinations of colors 0, 1, 2, and 3 of a two-plane-deep object into the selections of colors 0, 1, 4, and 5.

If, conversely, PlaneOnOff had a value of binary 1010, the color selections would be:

1 0 1 0 = color 10

1 0 1 1 = color 11

1 1 1 0 = color 14

1 1 1 1 = color 15

In other words, for this one, there won't be a transparent color in the bunch. PlanePick and PlaneOnOff are actually 8-bit binary values (UBYTE), for which only six bits are used in the current version of the Amiga hardware. You can experiment with these values to translate colors. Note that PlanePick and PlaneOnOff can be used to translate any depth of defined object into whatever depth of playfield is to be drawn.

The PurgeGels Routine

You will need two routines for returning memory to the system. One is called PurgeGels; it undoes what ReadyGels sets up. The other is called DeleteGels; it undoes what MakeBob and MakeVSprite do. DeleteGels assumes that all gels are attached to the system gel list. Listing 7.6 is the source code for PurgeGels and DeleteGels.

Advantages to Using Bobs

Bobs let you define objects that can be any width by any height, as long as you have the memory space to support them. Contrast this with sprites, which can be only 16 bits wide.

Bobs give you a greater variety of color choices than sprites. Instead of three possible colors plus transparent (or 15 colors plus transparent in a special attach mode not covered here), you can have as many colors in the body of a Bob as there are colors available in the playfield (screen) area. This can mean up to 4,096 colors in the special hold and modify mode.

For Bobs, you can specifically define the drawing priority by manipulating the Before and After pointers. You can tell the system to draw this Bob before that Bob and after this other one, thereby continuously maintaining the priority levels between Bobs that you specify.

Disadvantages to Using Bobs

Because Bobs are drawn as part of the playfield area, using Bobs is slower than using virtual sprites. In addition, because they are part of

```
/* purgegels.c */

/* Use this to get rid of the gels stuff when it is not needed any more.
 * You must have allocated the gels info stuff (use the ReadyGels routine). */

PurgeGels(g)
struct GelsInfo *g;
{
   if (g->collHandler != NULL)
      FreeMem(g->collHandler, sizeof(struct collTable));
   if (g->lastColor != NULL)
      FreeMem(g->lastColor, sizeof(LONG) * 8);
   if (g->nextLine != NULL)
      FreeMem(g->nextLine, sizeof(WORD) * 8);
   if (g->gelHead != NULL)
      FreeMem(g->gelHead, sizeof(struct VSprite));
   if (g->gelTail != NULL)
      FreeMem(g->gelTail, sizeof(struct VSprite));
}


/* Deallocate memory that has been allocated by the routines Makexxx. */
/* Assumes images and imageshadow deallocated elsewhere. */

DeleteGel(v)
struct VSprite *v;
{
   if (v != NULL) {
      if (v->VSBob != NULL) {
         if (v->VSBob->SaveBuffer != NULL) {
            FreeMem(v->VSBob->SaveBuffer, sizeof(SHORT) * v->Width
                  * v->Height * v->Depth);
         }
         if (v->VSBob->DBuffer != NULL) {
            if (v->VSBob->DBuffer->BufBuffer != 0) {
               FreeMem(v->VSBob->DBuffer->BufBuffer,
                     sizeof(SHORT) * v->Width * v->Height * v->Depth);
            }
            FreeMem(v->VSBob->DBuffer, sizeof(struct DBufPacket));
         }
         FreeMem( v->VSBob, sizeof(struct Bob));
      }
      if (v->CollMask != NULL) {
         FreeMem(v->CollMask, sizeof(WORD) * v->Height * v->Width);
      }
      if (v->BorderLine != NULL) {
         FreeMem(v->BorderLine, sizeof(WORD) * v->Width);
      }
      FreeMem(v, sizeof(struct VSprite));
   }
}

/* end of listing 7.6, purgegels.c */
```

Listing 7.6: The purgegels routine

the drawn background, it is often necessary (as is done in the program in Listing 7.7) to double-buffer the display. This means displaying one screen while drawing into a different screen, then swapping the screen positions and repeating the action.

Bobs use more memory than virtual sprites since it is usually appropriate to set SAVEBACK—save the background area that the Bob covers

up when drawn—so that the system can restore the background later when the Bob is moved. For every Bob that is on the screen, there is just as much extra memory space being used to save the background it overlaps.

The Bob Program

When you run the program in Listing 7.7, you will notice that drawing happens directly into the screen's drawing area rather than in a window.

The graphics animation capabilities of the Amiga are extensive. This chapter has provided you with tools that you can incorporate into dynamic graphics in your own programs.

```
/* bobdemo.c */

/* This example follows the vsprite example as closely as possible
 * to demonstrate that the gel system manages both VSprites and
 * Bobs.  By examining the two examples side by side, you can
 * see how Bobs and VSprites differ.  The same image data is
 * used in both examples.  You'll notice that Bobs have no
 * effect on the background display colors since they save
 * and restore their backgrounds, and have no effect on
 * the sprite color registers. */

/* (NOT double-buffered) */

#include "exec/types.h"
#include "intuition/intuition.h"
#include "graphics/sprite.h"
#include "exec/memory.h"
#include "graphics/gels.h"

/* ask system to create and manage MAXSP Bobs */
#define MAXSP 8

/* define possible speeds for Bobs in counts per vblank */

SHORT speed[] = { 1, 2, -1, -2 };

SHORT xmove[MAXSP], ymove[MAXSP];
                                /* sprite directions of movement */

/* changed for Bobs */
struct Bob *bob[MAXSP];         /* MAXSP Bobs */
struct VSprite *vspr;           /* pointer to a sprite */

/* added for Bobs */
/* All the combinations of two bits on within 5;
 * selects where to assign planes of the image
 * of the Bob to make color choices */

BYTE pick[] = { 0x03, 0x05, 0x09, 0x11, 0x06,
                0x0c, 0x18, 0x0A, 0x12, 0x14 };
```

Listing 7.7: The bob program

```
short maxgot;                   /* max # of Bobs we created */

struct GelsInfo mygelsinfo;     /* the window's RastPort needs one
                                 * of these in order to do VSprites */

struct Window *w;       /* pointer to a Window   */
struct RastPort *rp;    /* pointer to a Window's RastPort */
struct Screen *s;       /* pointer to a Screen */
struct ViewPort *vp;    /* pointer to a ViewPort */

struct RastPort *srp;   /* pointer to the Screen's RastPort */
struct Window *OpenWindow();
struct Screen *OpenScreen();
LONG GfxBase;
LONG IntuitionBase;

/* 18 words of Bob data = 9 lines of data, each, in two planes */

UWORD bob_data[ ] = {
                        /* first plane image ... equal to left word
                         * of the vsprite image data */

                        0x0fc3,
                        0x3ff3,
                        0x30c3,
                        0x0000,
                        0x0000,
                        0x0000,
                        0xc033,
                        0xffc0,
                        0x3f03,

                        /* second plane image ... equal to the right
                         * word of the vsprite image data. */

                        0x0000,
                        0x0000,
                        0x0000,
                        0x3c03,
                        0x3fc3,
                        0x03c3,
                        0xc033,
                        0xffc0,
                        0x3f03,
                        /* plane 3 of image (actually only
                         * 2 planes of real data, but
                         * InitMasks, as called in MakeBob
                         * and MakeVSprite requires these
                         * extra planes of zeros to make
                         * the mask correctly */

                        0,0,0,0,0,0,0,0,0,
                        /* plane 4 */
                        0,0,0,0,0,0,0,0,0,
                        /* plane 5 */
                        0,0,0,0,0,0,0,0,0 };

UWORD *bobdata;
UWORD *sprdata;

/* To move a Bob, you move its underlying vsprite, so the title
 * of the routine is still appropriate. */
movesprites()
{
```

Listing 7.7: The bob program (continued)

```
        short i;

        for (i=0; i<maxgot; i++)
        {
                vspr = bob[i]->BobVSprite;      /* changed for Bobs */

                vspr->X = xmove[i]+vspr->X;
                vspr->Y = ymove[i]+vspr->Y;

                /* move the sprites ... here. */

                if(vspr->X >= 300 || vspr->X <= 0) xmove[i]=-xmove[i];
                if(vspr->Y >= 190 || vspr->Y <= 0) ymove[i]=-ymove[i];

        }

        SortGList(srp);         /* get the list in order */
        DrawGList(srp, vp);     /* create the sprite instructions */

        MakeScreen(s);
        RethinkDisplay();

        return(0);
}


#define AZCOLOR 1
#define WHITECOLOR 2

#define WC WINDOWCLOSE
#define WS WINDOWSIZING
#define WDP WINDOWDEPTH
#define WDR WINDOWDRAG

#define NORMALFLAGS (WC|WS|WDP)
/* Did not use windowdrag because don't want screen to be moved. */

/* Allow window sizing so user can decrease size of window, then
 * increase it again, thus erasing the background text. User can
 * easily see that some vsprites wink out and into existence when
 * too many sprites are on a single horizontal plane and the
 * vsprite system runs out of sprites to assign.
 */

/* myfont1 specifies characteristics of the default font;
 * this case selects the 80-column font that displays as
 * 40 columns in low-resolution mode.
 */
struct TextAttr myfont1 = { "topaz.font", 8, 0, 0 };

struct NewScreen myscreen1 = {
        0, 0,           /* LeftEdge, TopEdge ... where to put screen */
        320, 200,       /* Width, Height ... size of the screen */
        5,              /* 5 planes Depth, means 2 to the 5th or
                        * 32 different colors to choose from once
                        * the screen is opened.
                        */
        1, 0,           /* DetailPen, BlockPen */
        SPRITES,        /* ViewModes ... value of 0 = low resolution */
        CUSTOMSCREEN,   /* Type of screen */
        &myfont1,       /* Font to be used as default for this screen */
        "32 Color Test", /* DefaultTitle for its title bar */
        NULL,           /* screens user-gadgets, always NULL, ignored */
        NULL };
```

Listing 7.7: The bob program (continued)

```
                        /* address of custom bitmap for screen,
                         * not used in this example
                         */

struct NewWindow myWindow = {
        0,              /* LeftEdge for window measured in pixels,
                           at the current horizontal resolution,
                           from the leftmost edge of the Screen */
        0,              /* TopEdge for window measured in lines
                           from the top of the current Screen.   */
        320, 185,       /* Width, Height of this window */
        0,              /* DetailPen - what pen number is to be
                           used to draw the borders of the window */
        1,              /* BlockPen - what pen number is to be
                           used to draw system generated window
                           gadgets */
                        /* (for DetailPen and BlockPen, the value
                            of -1 says "use the default value")  */
        CLOSEWINDOW,    /* simplesprite program used INTUITICKS also */
                        /* IDCMP Flags */
        NORMALFLAGS | GIMMEZEROZERO | ACTIVATE | BACKDROP,
                        /* Window Flags */
        NULL,           /* FirstGadget */
        NULL,           /* CheckMark   */
        "Click Close Gadget To Stop",   /* Window title */
        NULL,           /* Pointer to Screen if not Workbench */
        NULL,           /* Pointer to BitMap if a SUPERBITMAP window */
        320, 10,        /* minimum width, minimum height */
        320, 200,       /* maximum width, maximum height */
        CUSTOMSCREEN
        };

#include "graphics/gfxmacros.h"

/* #include "event1.c"  */
/* gets the event handler */
/* event1.c */

HandleEvent(code)
        LONG code;      /* provided by main */
{
        switch(code)
        {
        case CLOSEWINDOW:
                return(0);
                break;
        case INTUITICKS:
                movesprites();  /* 10 moves per second; test */
                default:
                        break;
        }
return(1);
}

UWORD mycolortable[] = {

        0x0000, 0x0e30, 0x0fff, 0x0b40, 0x0fb0, 0x0bf0,
        0x05d0, 0x0ed0, 0x07df, 0x069f, 0x0c0e,
        0x0f2e, 0x0feb, 0x0c98, 0x0bbb, 0x07df,

        0x0000, 0x0e30, 0x0fff, 0x0b40,
        0x0fb0, 0x0bf0, 0x05d0, 0x0ed0,
        0x07df, 0x069f, 0x0c0e, 0x0f2e,
```

Listing 7.7: The bob program (continued)

```
        0x0feb, 0x0c98, 0x0bbb, 0x07df
        };

        /* black, red, white, fire-engine red, orange, yellow,
        lime green, green, aqua, dark blue, purple,
        violet, tan, brown, gray, skyblue, (everything again) */
                .
UWORD colorset0[ ] = { 0x0e30, 0xffff, 0x0b40 }; /* same as colors 17-19 */
UWORD colorset1[ ] = { 0x0bf0, 0x05d0, 0x0ed0 }; /* 21-23 */
UWORD colorset2[ ] = { 0x069f, 0x0c0e, 0x0f2e }; /* 25-27 */
UWORD colorset3[ ] = { 0x0c98, 0x0bbb, 0x07df }; /* 29-31 */
UWORD *colorset[ ] = {
                        colorset0, colorset1,
                        colorset2, colorset3 };
int choice;
char *numbers[] = { "17","18","19",
                    "20","21","22","23",
                    "24","25","26","27",
                    "28","29","30","31" };

#include "ram:purgegels.c"
#include "ram:readygels.c"
#include "ram:makevsprite.c"
/* added for Bobs */
#include "ram:makebob.c"
main()
{
        struct IntuiMessage *msg;
        LONG result;
        SHORT k, j, x, y, m, error;
        UWORD *src, *dest;      /* for copying sprite data to RAM */

        GfxBase = OpenLibrary("graphics.library",0);

        IntuitionBase = OpenLibrary("intuition.library",0);
        /* (error checking left out for brevity here) */

        s = OpenScreen(&myscreen1);  /* try to open it */
        if(s == 0)
        {
                printf("Can't open myscreen1\n");
                exit(10);
        }
        myWindow.Screen = s;    /* say where screen is located */

        ShowTitle(s, FALSE);    /* Don't let screen be dragged down...*/

        w = OpenWindow(&myWindow);
        if(w == 0)
        {
                printf("Window didn't open!\n");
                CloseScreen(s);
                exit(20);
        }
        vp = &(s->ViewPort);
        /* set the colors for this viewport */

        srp = &(s->RastPort); /* drawing Bobs directly into Screen */

        LoadRGB4(vp, &mycolortable[0], 32);
        rp = w->RPort;
                /* Now wait for a message to arrive from Intuition
                 * (task goes to sleep while waiting for the message)
                 */
```

Listing 7.7: The bob program (continued)

```
        /* Write Text using sprite colors so that demo can show
         * how the vsprite system stuffs the colors as it goes
         * down the screen.  Thus if using vsprites, user should
         * avoid using the color registers that the vsprites use.
         * Bobs don't bother any of the color registers.        */


        for(j=8; j<180; j+=50)
        {
                for(k=0; k<15; k++)
                {
                        Move(rp,k*20,j);
                        SetAPen(rp,k+17);       /* show 17-31 */
                        /* 16, 20, 24, 28 are unaffected by vsprites
                         * because they are not used by hardware sprites */

                        Text(rp,numbers[k],2);
                }
        }
/*
        ScreenToBack(s);
*/
        /* ************************************* */
        /* BOB DEMO SECTION                      */
        /* ************************************* */

        /* Allocate CHIP memory to hold the actual Bob data */
        /* (necessary if ever to run on an expanded RAM Amiga) */

        bobdata = (UWORD *)AllocMem(90, MEMF_CHIP);

        if(bobdata == NULL)
        {
                /* not enough memory for Bob */
                printf("No memory for bobdata!\n");
                goto finish;
        }
        /* now copy the sprite data into the CHIP RAM. */

        src = bob_data;
        dest = bobdata;         /* source, destination */

        for( j=0; j<45; j++)
        {
                *dest++ = *src++;
        }

        maxgot = 0;

        /* Prepare the GELS system to work with VSPRITE or Bobs */

        error = ReadyGels(&mygelsinfo, srp);

        for(k=0; k<MAXSP; k++)  /* whatever maximum number of BOBS */
        {
                xmove[k]=speed[RangeRand(4)];
                ymove[k]=speed[RangeRand(4)];

                /* establish a position for the BOB */
                x = 10 + RangeRand(280);
                y = 10 + RangeRand(170);

                /* Demonstrate that Bobs give a wider variety of color
```

Listing 7.7: The bob program (continued)

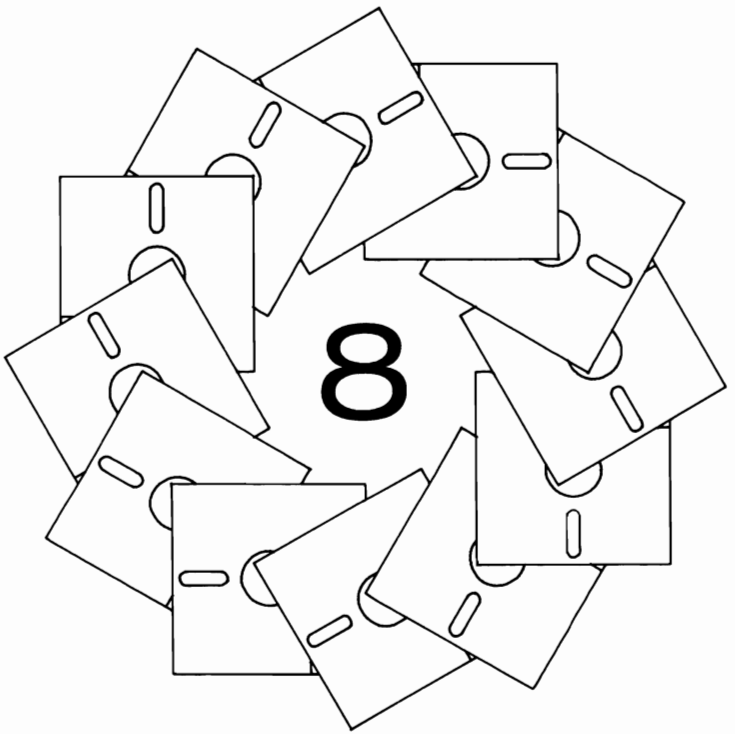
```
                * choices by using planepick and planeonoff to select
                * colors other than those available by simply using
                * vsprites. */

               /* create a Bob */
               bob[k] = MakeBob( 16, 9, 5, bobdata,
                        pick[RangeRand(10)], RangeRand(31),
                                 x,y,SAVEBACK | OVERLAY );

               /* 16 bits wide, 9 lines tall, 5 planes deep,
                * (even though the image itself is only 2 planes
                * deep, there are 5 planes worth of data to be
                * saved for each Bob we draw!!!!!!!).  Bobdata
                * in chip memory, position the two planes at any 2
                * out of 5 bitplanes, and then arrange a random
                * filling of either 1's or 0's in the nonpicked
                * planes so as to create the appropriate color
                * in the bit mask of the bob object; put at x,y;
                * save and restore the background as the Bob moves */

               if(bob[k] == 0)
               {
                       printf("Ran out of memory during makebob\n");
                       goto finish;
               }
               if(k > 0)
               {
                       /* establish a definite drawing order */
                       m = k-1;
                       bob[k]->After = bob[m]->Before;
                       bob[k]->After->Before = bob[k];
               }

               AddBob(bob[k], srp);

               maxgot++;

               /* Bobs have nothing to do with colorsets. */
       }

       while(1)        /* forever */
       {
               WaitTOF();
               movesprites();

               result = -1;    /* now see if CLOSEWINDOW is waiting */
               msg = (struct IntuiMessage *)GetMsg(w->UserPort);
               if(msg != 0)
               {
                       result = HandleEvent(msg->Class);

                       /* Let Intuition reuse the msg */
                       ReplyMsg(msg);
               }
               if(result == 0)
               {
                       break;  /* got a CLOSEWINDOW */
               }
       }
               /* DONE, now cleanup */

               /* Free however many Bobs we actually managed to create */
       finish:
```

Listing 7.7: The bob program (continued)

```
        for(k=0; k<maxgot; k++)
        {
                if(bob[k])
                {
                        DeleteGel(bob[k]->BobVSprite);
                }
        }
        /* delete what ReadyGels created */

        PurgeGels(&mygelsinfo);
        if(bobdata)
        {
                FreeMem(bobdata, 90);   /* free what we allocated. */
        }
        if(w)
        {
                CloseWindow(w);
        }
        if(s)
        {
                CloseScreen(s);
        }
}
```

Listing 7.7: The bob program (continued)

Sound



This chapter introduces the sound system on the Amiga. It shows you how to reserve audio channels for your own use and how to store data directly in the audio registers. To understand the purpose of the various audio functions that the Amiga sound software provides, it is helpful to first understand something about the audio hardware.

AUDIO HARDWARE

For most Amiga functions, such as graphics, there are several layers of system software between your program and the Amiga hardware. The audio software system provides considerably less software overhead and expects that the programmer of the sound system will often directly manipulate the sound hardware to get the desired result.

The sound hardware on the Amiga consists of four independent audio channels, two of which (channels 0 and 3) are connected to the left stereo output jack on the back of the Amiga, and two of which (channels 1 and 2) are connected to the right jack. Each channel has a volume control register that can set the volume for that channel to one of 64 possible unique values.

Each channel has a period register and a data register. The period register is used to establish the sampling rate for the channel. As higher and higher values are installed in this period register, a lower and lower sampling rate is established and a lower frequency (a lower note) is output. The data register points to the place where the data for that channel is stored in chip-accessible memory (MEMF_CHIP). In addition, a length register defines the length of the memory space in which the sound data resides.

Each of the four audio channels can retrieve its data via Direct Memory Access (DMA) or directly from the processor. DMA is the most common method. Using DMA means that the audio system can retrieve data automatically while the processor is doing something else. With a total of 26 DMA channels on the Amiga, including the four DMA channels dedicated to the hardware, your program can go on to do other things, such as calculate the next audio output waveform or retrieve data from the user while the current sound is being output, without intervention of the 68000.

To produce data that will result in a sound output from the Amiga, you provide an area of memory in which you have built a numerical representation of the audio output data. If you are operating exclusively with the hardware, after filling the data area, you set the volume to the level you wish to use, establish the period register value to set the rate of data

output to the audio tone generation circuitry, point the data register to the memory area where your data is stored, write the length of the data into the length register, and start the audio DMA.

COMMUNICATING WITH THE AUDIO DEVICE

As with other devices, the Audio device uses an IORequest block for communicating with your task. The request block for audio is called IOAudio. To gain access to the Audio device, you use the OpenDevice function, passing it an IOAudio structure that contains all zeros. OpenDevice will initialize the address of the Audio device (io_Device), which will be needed later when you call any of the Audio device's functions.

Listing 8.1 is some sample code that shows how you begin access to the Audio device. It provides a subroutine that you can use to dynamically allocate an IOAudio data structure, initialized to all zeros. You can use this data structure to create IOAudio request blocks for use with the audio routines. The CreateAudioIO routine returns a pointer to an IOAudio request block. You complete the initialization of other data fields for your own use. A corresponding routine, also shown, frees the memory that this IOAudio request block uses. Your program should keep track of how many of these blocks it creates, and it should free all of them when it exits.

```
/* newaudio.c */

struct IOAudio *
CreateAudioIO()
{
        struct IOAudio *iob;

        iob = (struct IOAudio *)
              AllocMem(sizeof(struct IOAudio),
                       MEMF_CHIP | MEMF_CLEAR);

        return(iob);    /* returns 0 if out of memory */
}

void
FreeAudioIO(iob)
struct IOAudio *iob;
{
        FreeMem(iob, sizeof(struct IOAudio));
}
```

Listing 8.1: The newaudio routines

To open the Audio device, you use a pointer to an IOAudio request block that you have allocated. Listing 8.2 shows a sample function call for opening the Audio device.

The global variable auDevice is set to the value returned in the io_Device field from the OpenDevice call so that you can copy the value to other uninitialized request blocks for additional audio function calls.

When an allocation has been completed, the allocation key value is generated and stored with the audio channel. If your future requests do not match the allocation key (ioa_AllocKey), your command will be rejected. Thus, the allocation key that is returned from the OpenDevice function is stored for future use.

AUDIO SOFTWARE

The audio software provides ways of adapting the Amiga hardware to the needs of a multitasking system. The software includes routines for the following:

- Allocating one or more channels for the exclusive use of a particular task.
- Establishing a priority of use to allow a more important sound to be heard, even if another task already has a channel allocated.
- Letting a lower priority task finish using a channel before it is stolen for another task's use.
- Enabling you to start or stop an audio channel as well as to queue several audio sounds for a channel to play in sequence.
- Specifying how often a particular waveform definition should be played as well as informing you about which waveforms have been played or are currently playing.
- Enabling you to go directly to the hardware if you wish to exercise that level of control.

Allocating Channels

In the Amiga multitasking system, you might want to get the use of one or more audio channels but also allow your use of the channels to be preempted (allowing another task to steal a channel, or simply to request the use of a channel).

The audio system lets you request channels to be reserved for your task by two different methods: during an OpenDevice function call, and

```
int error;
struct Device *auDevice;
struct IOAudio *audioIOB;
WORD allocationKey;

audioIOB = CreateAudioIO();

/* If audioIOB return value is 0, you ran out */
/* of memory and should exit showing an error. */
error = OpenDevice("audio.device",0,audioIOB,0);

/* If error is not 0, again, should exit */

/* Get the device address for later use */

auDevice = audioIOB->ioa_Request.io_Device;

/* Get the allocation key for later use */

allocationKey = audioIOB->ioa_AllocKey;

/* Now the rest of the audio request block can be */
/* initialized and it can be sent to the device */
/* using SendIO or DoIO (or BeginIO, which is, in */
/* effect, a faster version of SendIO) */
```

Listing 8.2: Opening the Audio device

as a separate function, called ADCMD_ALLOCATE. During either of these two possible allocation times, the priority of your task's use of the channels you request is established by the priority field of the IOAudio request block, ioa_Request.io_Message.mn_Node.ln_Pri. If you use the CreateAudioIO routine, the priority value for your audio channel request will be set to zero. Here is an example that sets the priority to 127, where iob is a pointer to an IOAudio request block:

```
struct AudioIO *iob;

/* set the priority for this request block to 127 */

iob->ioa_Request.io_Message.mn_Node.ln_Pri = 127;
```

Priority values can be set from − 128 (minimum) to + 127 (maximum). If you set the priority value to maximum, the system will simply refuse to abort your task's use of the channel. If you set a lower priority value and another task requests allocation using a higher value in its allocation request, either your task's I/O request will be aborted, or your task will be told that another task wants to use one or more of its channels on a priority basis.

The priority of the channel is established when the OpenDevice function is performed. If you wish to change the priority value later, you

Audio Channel	Bit Position in Data Byte	Binary Value	Decimal Value
0	0	0001	1
1	1	0010	2
2	2	0100	4
3	3	1000	8

Table 8.1: Audio channel bit positions and values

must use the ADCMD_SETPREC command, shown later in this chapter.

Regardless of when you request use of one or more channels (at OpenDevice or by a separate call to ADCMD_ALLOCATE), your request takes the same form: pointing your io_Data pointer to the first byte in an array of allocation request bytes, and setting the ioa_Length value in your request block to specify how many bytes there are in that array.

Each allocation byte contains four binary bits (the least significant four bits of the byte) that tell the system which channel(s) of the four possible audio channels you want to reserve for your own use. The bit positions correspond directly to the audio channel numbers, starting with channel 0, as shown in Table 8.1.

To reserve two channels for your task to use, you form a value called an allocation mask, which is composed of values representing one left channel and one right channel.

For your allocation mask, you can be very specific and say "If I can't have channels 0 and 1, I don't want any channels at all." Or, you can ask the system to search several combinations for you in a single allocation call, such as "Give me channels 0 and 1, or 0 and 2, or 3 and 1, or 3 and 2." These combinations are those that yield a stereo pair. To make this request, you provide a four-byte array, with each byte representing one of the four bit combinations that form a stereo pair, as shown in Table 8.2.

When the system completes your allocation request, the channels that you now have for your task's exclusive use are indicated by the io_Unit field of your I/O request. Listing 8.3 utilizes the command ADCMD_ALLOCATE (after the device is open) to attempt to allocate either a single channel or a stereo pair. Each routine in the listing returns a value that indicates which channels have been allocated for your task to use.

Note that once the system allocates a channel for your use, your task must free that channel later. If the channel is not freed, no other task will be allowed to use that channel until the next system reset.

Channels	Byte Value (binary)	Byte Value (decimal)
0 and 1	0011	3
0 and 2	0101	5
3 and 1	1010	10
3 and 2	1100	12

Table 8.2: Stereo audio channel allocation values

When the system allocates one or more channels in a single request, it also provides a value that you use as a key to access the channel. It is possible for any of the four audio channels to queue one or more requests for audio output. As each request reaches the head of the queue, the value of ioa_AllocKey in the request is matched against the key value that the Audio device keeps internally for each channel. It is by this key value that the channel knows which task is its owner.

Each time a channel is freed, then reallocated, a new key value is generated. Requests that have the correct key value are performed; requests that have an incorrect value are returned to the sender, indicating an error. Thus, the routines in Listing 8.3 require an initialized IOAudio request block (just the io_Device and mn_ReplyPort fields), and the address at which a global word variable is located, into which the allocation key can be stored. All control functions that you exercise on the audio channels require that this allocation key be correct within the IOAudio request block or the command will be rejected.

If you wish, you can also utilize OpenDevice to automatically allocate channels as the device opens. This method requires the same setup as in the last listing, but it does not require a separate call to ADCMD_ALLOCATE. Listing 8.4 is a routine that allocates during OpenDevice and returns the value representing which channel was allocated.

Note that once the device is opened, you can retrieve a pointer to it from the io_Device field of the IOAudio request block and the allocation key from the ioa_AllocKey field of the request block. Also note that the value in the io_Unit field of the request block contains the mask value that defines which channels have been allocated with this request.

It is not necessary to open the device several times. To get another channel once the device is opened, you should use the ADCMD_ALLOCATE command.

```
/* getaudio.c */

WORD mykey;             /* a global value; key for access */

UBYTE
GetAnyStereoPair(iob)
struct IOAudio *iob;
{
        /* Assume that iob already initialized
         * in Device field */

        /* Also assume that the ReplyPort
         * field is initialized.  */

        UBYTE stereostuff[4];
        UBYTE mychan;
        int error;

        stereostuff[0] = 3;
        stereostuff[1] = 5;
        stereostuff[2] = 10;
        stereostuff[3] = 12;

        /* Set the precedence of the channel request */

        iob->ioa_Request.io_Message.mn_Node.ln_Pri = 20;

        /* Type of command is allocate */

        iob->ioa_Request.io_Command = ADCMD_ALLOCATE;

        /* Point to the allocation map */

        iob->ioa_Data = (UBYTE *)stereostuff;

        /* It contains 4 entries */

        iob->ioa_Length = 4;

        /* Don't wait for allocation, channels
         * should be available!  If we don't set
         * ADIOF_NOWAIT, the task will idle waiting
         * for a chance to allocate the channel,
         * looking again each time another task
         * allocates or frees a channel.          */

        iob->ioa_Request.io_Flags = ADIOF_NOWAIT | IOF_QUICK;

        BeginIO(iob);

        error = WaitIO(iob);  /* returns nonzero if error */
        if(!(iob->ioa_Request.io_Flags & IOF_QUICK))
        {
                /* if flag not set, then the message
                 * was appended to the reply port
                 * (was not quick I/O after all)  */
                GetMsg(iob->ioa_Request.io_Message.mn_ReplyPort);
        }
        if(error)
        {
                return(0);
        }
        mychan = ((ULONG)(iob->ioa_Request.io_Unit)) & 0xFF;
        return(mychan);
}
```

Listing 8.3: The getaudio routines

```
UBYTE
GetAnyChannel(iob)
struct IOAudio *iob;
{
        UBYTE anychan[4];
        UBYTE mychan;
        int error;

        anychan[0] = 1;
        anychan[1] = 2;
        anychan[2] = 4;
        anychan[3] = 8;

        iob->ioa_Request.io_Message.mn_Node.ln_Pri = 20;
        iob->ioa_Request.io_Command = ADCMD_ALLOCATE;
        iob->ioa_Data = (UBYTE *)anychan;
        iob->ioa_Length = 4;
        iob->ioa_Request.io_Flags = ADIOF_NOWAIT | IOF_QUICK;
        BeginIO(iob);
        error = WaitIO(iob);  /* returns nonzero if error */

        if(!(iob->ioa_Request.io_Flags & IOF_QUICK))
        {
                GetMsg(iob->ioa_Request.io_Message.mn_ReplyPort);
        }
        if(error)
        {
                return(0);
        }
        mychan = ((ULONG)(iob->ioa_Request.io_Unit)) & 0xFF;
        return(mychan);
}
```

Listing 8.3: The getaudio routines (continued)

Waiting for Allocation

In all forms of allocation, your task is put to sleep until one of your allocation requests has been fulfilled. Each time one or more channels has been freed, the Audio device will examine the allocation requests and try to fulfill the request that has the highest priority. If there are still not enough channels free to fulfill this request, the task will continue to sleep awaiting the reply from the Audio device.

Waiting for allocation (using DoIO) is demonstrated in the GetAnyChannel and GetAnyStereoPair routines. Although it is not visible in the OpenAnyAudio routine, the task does not return from the OpenDevice call until the allocation succeeds.

If you do not want your task to wait for allocation, then you can set an additional flag, ioa_Flags, to a value of ADIOF_NOWAIT. If this flag contains other than a zero and the allocation cannot be performed immediately, the request block will be returned immediately. The error value (ioa_Request.io_Error) will be IOERR_ALLOCFAILED. If the allocation

```
/* openanyaudio.c */

BYTE
OpenAnyAudio(iob)
struct IOAudio *iob;
{
        int error;
        BYTE anychan[4];
        BYTE mychannel;

        anychan[0] = 1;
        anychan[1] = 2;
        anychan[2] = 4;
        anychan[3] = 8;

        iob->ioa_Data = (UBYTE *)anychan;
        iob->ioa_Length = 4;

        error = OpenDevice("audio.device",0,iob,0);

        /* If error not 0, should return 0 for key */
        /* and for unit value */

        if(error != 0)
        {
                return((BYTE)0);
        }
        else
        {
                return(((ULONG)(iob->ioa_Request.io_Unit)) & 0xFF);
        }
}
```

Listing 8.4: The openanyaudio routine

fails, the value returned from any of the allocation routines will be zero, and the error value will indicate the failure code IOERR_ALLOCFAILED.

Here is an example that shows setting the flag for immediate return:

```
iob->ioa_Request.io_Flags = ADIOF_NOWAIT;
```

Using the Allocated Channels

When you open the Audio device, it fills in the ioa_Request.io_Device field of the IOAudio request block that you provided to the OpenDevice function.

When you allocate one or more channels, whether during OpenDevice on the Audio device, or later by using ADCMD_ALLOCATE, the system fills in the ioa_Request.io_Unit field of your request with the value representing the channel or channels you have allocated. The system also assigns a unique allocation key in ioa_AllocKey representing this request.

If you use CreateAudioIO to create more than one IOAudio request block so as to queue up multiple audio commands, then all three values—the device, the unit, and the allocation key—must be copied from an

initialized IOAudio request block in order to be used to send requests to that channel.

The only exception to this is when you have successfully allocated a pair of stereo channels for your task. In this case, you change the unit value so that you send commands separately to each of the stereo channels you have allocated. For example, if you have successfully allocated channels 1 and 2, the unit value that you receive will be binary 0110. You will send commands to the channels individually using unit values of 0100 (for channel 2) and 0010 (for channel 1). Your allocation request, returning the original value of 0110 binary, simply indicated that you obtained use of both channels 1 and 2. It is then up to you to send them commands individually by splitting the unit number value that you have received.

If you want your task to go on to do something else if an audio allocation request fails the first time it is attempted, set the command flag variable to include the value ADIOF_NOWAIT.

Locking a Channel

The priority value that you set on your audio request directs the Audio device to reject requests that have an equal or lower priority than yours. If you want to keep a channel forever, set your priority value to 127. Once you are granted use of that channel, no other task's request will be honored until you free that channel. In this case, you will not need to lock and unlock the channel. However, if your task wants to share the system, you will want to use a lock.

If another task makes an allocation request that has a higher priority, you have two choices: let the channel be stolen from your task, with your output aborted in midstream, or establish a lock on the channel. If you establish a lock, your task receives a message stating that another higher priority task wishes to use the channel and that it should finish whatever it needs to do and free the channel as soon as possible. If you are simply using the standard audio device I/O commands, locking a channel is not necessary.

However, if, after allocating a channel, your task is writing directly to the audio registers, you should use the locking function to clean up appropriately, and to explicitly stop using the registers. This frees the channel and allows the higher priority task access to it. If you do not perform the lock function before writing directly to the registers, both your task and this other task will be attempting to use the same output registers, with unpredictable results.

Generally, you will want to lock a channel immediately after it has been allocated for your use. If the lock fails, it means that another task with a

higher priority value than yours has already stolen the channel from you. If the lock succeeds, you have exclusive use of that channel until you free it, regardless of the priority value of the channel.

To use Listing 8.5, the IOAudio request block must contain the allocation key and the auDevice value. To perform I/O, you also need a reply port to which the I/O request can be sent when the I/O is completed. Thus, the ioa_Request.ReplyPort must contain the valid address of the reply port.

Note that the IOAudio request block that is used in this example is simply sent to the Audio device and is held (not returned to your task) until either another task tries to steal the channel or this task frees the channel. Thus, you can see yet another reason for using the CreateAudioIO function: it provides multiple request blocks to use for communicating with the Audio device.

Setting a New Priority Value

You might be producing several sounds, some of which are important and some of which are background noises. For the important sounds,

```
/* lockchan.c */

LockAChannel(lockiob,channel)
struct IOAudio *lockiob;
UBYTE channel;
{
        /* tell it which channel to lock */

        lockiob->ioa_Request.io_Unit = (struct Unit *)channel;

        lockiob->ioa_Request.io_Command = ADCMD_LOCK;
        lockiob->ioa_Request.io_Flags = 0;

        /* Send this command.  It does not return to
         * the reply port unless and until either this
         * task frees the channel or another channel of
         * higher precedence steals it.  Appropriate
         * to keep two reply ports, perhaps... one for
         * standard I/O replies, another for channel steal
         * requests.
         */
        BeginIO(lockiob);
}

FreeAChannel(iob)
struct IOAudio *iob;
{
        /* allocation key and unit number must already be valid */
        iob->ioa_Request.io_Command = ADCMD_FREE;
        iob->ioa_Request.io_Flags = IOF_QUICK;
        BeginIO(iob);
        WaitIO(iob);
}
```

Listing 8.5: The lockchan routine

you will not want other tasks to steal your audio channels. Thus, you will want a high priority value on your audio channel while the important sounds are being output and a low priority value on the channel when less important sounds are being output.

The ADCMD_SETPREC command lets you dynamically establish the precedence (priority value) of a channel that you own. Listing 8.6 is a routine for setting a new precedence value. Note that the current precedence value will have been set during the OpenDevice function.

Controlling Audio Output

Now that you know how to allocate and free audio channels, you need to know how to provide data for the audio channels to output. There are a number of commands for controlling audio output. In addition, there are two commands that let you synchronize with the output of the audio channel. The control commands are the following:

CMD_WRITE	Queues up a waveform definition for the channel to output. This command can include the settings for period and volume of a waveform or can cause the period and volume values from the preceding CMD_WRITE to be used.
ADCMD_PERVOL	Changes the period or volume (or both) for a waveform that is already playing in a channel. The change can be immediate or can wait until the current waveform cycle has finished playing.

```
/* setprec.c */

int
SetChanPrecedence(iob,channel,prec)
struct IOAudio *iob;
BYTE channel;
BYTE prec;
{
        iob->ioa_Request.io_Command = ADCMD_SETPREC;
        iob->ioa_Request.io_Unit    = channel;
        iob->ioa_Request.io_Message.mn_Node.ln_Pri = prec;
        BeginIO(iob);
        WaitIO(iob);    /* wait for it to happen */
        return(iob->ioa_Request.io_Error); /* 0 if no error */
}
```

Listing 8.6: The setprec routine

ADCMD_FINISH	Stops the current CMD_WRITE waveform from playing, either immediately, or after the completion of the current cycle. The flag value ADIOF_SYNCCYCLE serves both ADCMD_PERVOL and ADCMD_FINISH, specifying whether to stop immediately (the default) or to stop only after completion of the current cycle (if the flag is set).
CMD_STOP	Stops the channel's output temporarily.
CMD_START	Restarts the channel's output.
CMD_FLUSH	Removes all current writes from the channel's queue.
CMD_RESET	Restores all audio registers to their initial state. Issues a CMD_FLUSH, issues a CMD_START, and restores audio interrupt vectors to system defaults. This is a multichannel command; it affects all audio hardware but has no effect on channel locking. The task that locked a channel must still free it.

The synchronization commands are as follows:

CMD_READ	Returns a pointer to the IOAudio request block, informing you that a particular audio channel is now playing. (It may be anywhere within the cycle.)
ADCMD_WAITCYCLE	Prevents the command from returning until the current CMD_WRITE has completed. (You use DoIO to send this command to a device.)

By using a combination of CMD_READ and ADCMD_WAITCYCLE, your task can synchronize with the audio output that it has queued. This feature can be useful for synchronizing sound and graphics action. For example, you may want to make a sound when an object hits and bounces off a wall.

Audio Data

To understand how to create audio data suitable for the audio channels, you need to know a few things about how the hardware interprets that data. To define audio output, you need to define three things:

- the waveform itself
- the sampling period
- the volume value

Waveform Definition

The waveform definition that you provide is a table of digital values that corresponds to the audio waveform you are trying to produce. Suppose that your waveform appeared as shown in Figure 8.1.

To produce this waveform, your data table must contain an even number of signed byte values representing the waveform value sampled at discrete, evenly spaced intervals. Generally speaking, you will get the best quality sound if you use the full dynamic range of the audio hardware. This means that, as shown in Table 8.3, you should draw a waveform whose top is at or near a value of + 127 and whose bottom is at or near − 127. This way, you provide the maximum number of possible steps at which the waveform can be measured. The process of

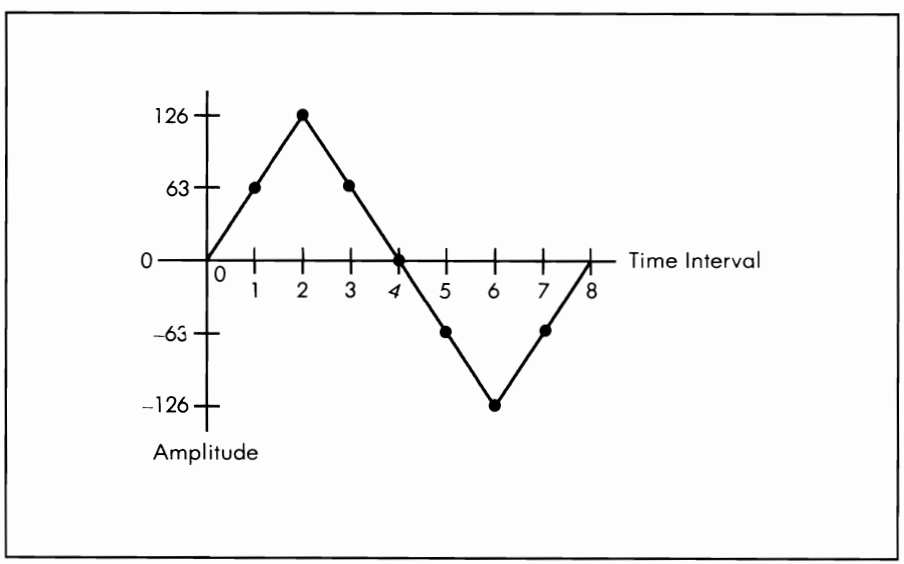


Figure 8.1: An audio waveform

converting a waveform into discrete digital step values is called analog-to-digital conversion.

Sampling Period

To construct the waveform from the digital data, the sound hardware sends the data to a digital-to-analog converter, one data item per sampling period. The audio circuitry, in turn, fills in the line segments between the discrete data samples, reconstructing the waveform.

The faster the sampling period, the more often a waveform table is sampled and, therefore, the higher the repetition rate of the system when reading the table. This means a higher frequency of output, or in other words, a higher note. The slower the sampling period, the lower the frequency of output, and the lower the note that is produced.

The sampling period for Amiga waveforms is specified as a divider value (called the period) rather than directly as a sampling rate. The divider value is applied to an internal sampling rate clock which, when divided by this value, determines the actual sampling rate. This means that the value that specifies the output frequency of a waveform is inversely proportional to the period value that controls the sampling rate. In other words, if you want a very high note, you use a very low period value. If you want a very low note, choose a high period value.

The minimum period value is 126 and is a limitation of the Amiga hardware. This limit is established by the design of the DMA hardware. See the *Amiga Hardware Manual* if you need more information about the limits of the audio hardware.

Time Interval Number	Amplitude of Waveform
0	0
1	63
2	126
3	63
4	0
5	-63
6	-126
7	-63

Table 8.3: Data for Figure 8.1

Clear Audio Output

As part of a CMD_WRITE, you can tell the system how many times a waveform is to be output. Thus, the same digital data table might be read many, many times before a new CMD_WRITE is issued. To avoid clicks and pops in audio output, you should define a waveform so that there is a smooth transition between waveforms if the same one is repeated. Of course, this also holds true for transition points between two different waveforms.

The smoothest transition between two types of audio waveforms occurs at zero crossings. That is, if your waveform starts and ends at a value of zero, it will be easy to splice together different waveform types since both touch the zero axis as each waveform ends and another begins.

If both waveforms begin and end, for example, at a value of 37, there will not necessarily be a distortion generated as long as the slope of the waveforms is the same at the splice point (if the first wave is going negative and the spliced wave is also going negative at very nearly the same rate). However, splicing at zero crossings is best because the voltage value to the audio output is zero then and the minimum amount of audio energy is available, thus avoiding a click when the waveform changes.

Positioning the Audio Data

Your audio data must be located in memory that is accessible to the custom chips. This means that you should allocate memory of the type MEMF_CHIP and copy or generate your data into that space. Memory outside the range of the custom chips cannot be read by the audio hardware.

The audio data must be a multiple of two bytes since the audio hardware retrieves two bytes each time any audio DMA takes place. In addition, the first byte must be located on an even byte boundary. You need not worry about the byte alignment if you use AllocMem, which automatically aligns its memory allocations on even byte boundaries.

Here is the code to allocate memory for the audio waveform shown in Figure 8.1:

```
BYTE *audata;

audata = (BYTE *)AllocMem(8,MEMF_CHIP); /* 8 bytes needed */
```

If audata is nonzero, the audio data can be copied into that area and used by the custom chips to produce an audio output.

Volume Control

By setting up the data table for the audio output, you have defined the appearance of the waveform itself. You must also specify how loud

the sound is to be. Each channel has a volume control associated with it; a value of 64 is the maximum volume; a value of 0 is the minimum volume. Values in between linearly control the volume of output of that channel.

Alternatively, the volume value can be set to maximum (as in the program that follows) and an external amplifier can be used to control the true volume of the output sound.

The Audio Program

Listing 8.7 is a full audio program that uses all four audio output channels to play the sample waveform at four different frequencies simultaneously. This program stores information directly in the audio registers. Thus, it also locks the channels before it tries to use them.

```
/* audio.c */

#include "exec/types.h"
#include "exec/memory.h"
#include "devices/audio.h"

extern struct IOAudio *CreateAudio(); /* fwd declaration */

BYTE trianglewave[8] = { 0, 63, 126, 63, 0, -63, -126, -63 };

UWORD period[4] = { 508, 428, 339, 254 };

struct Device *auDevice=0;
BYTE *chipaudio;
struct IOAudio *audioIOB[4];
struct IOAudio *aulockIOB[4];
struct IOAudio *aufreeIOB[4];

extern struct MsgPort *CreatePort();
struct MsgPort *auReplyPort, *auLockPort;
extern struct IOAudio *CreateAudioIO();
extern UBYTE GetAnyChannel();
extern void FreeAudioIO();

main()
{
    int error,i,chan;
    BYTE *s, *d;
    struct Unit *auunit;

    for(i=0; i<4; i++)
    {
            audioIOB[i]  = CreateAudioIO();
            aulockIOB[i] = CreateAudioIO();
            aufreeIOB[i] = CreateAudioIO();

            if(audioIOB[i] == 0 || aufreeIOB[i] == 0 | aulockIOB[i] == 0)
            {
                    finishup("out of memory!");
            }
```

Listing 8.7: The audio program

```
}

chipaudio = (BYTE *)AllocMem(8, MEMF_CHIP);
if(chipaudio == 0)
{
        finishup ("out of memory!");
}
d = chipaudio;  s = trianglewave;

for(i=0; i<8; i++)
{
        *d++ = *s++;     /* copy into chip memory */
}

error = OpenDevice("audio.device",0,audioIOB[0],0);

if(error)
{
        finishup ("audio device won't open!");
}
/* Get the device address for later use */

auDevice = audioIOB[0]->ioa_Request.io_Device;

/* Create ports for replies from the device */

auReplyPort = CreatePort(0,0);
auLockPort  = CreatePoequest.io_Device;

/* Create ports for replies from the device */

auReplyPort = CreatePort(0,0);
auLockPort  = CreatePort(0,0);

if(auReplyPort == 0 || auLockPort == 0)
{
        finishup("cannot create a port!");
}

for(i=0; i<4; i++)
{
        /* initialize port addresses and device fields
         * for all audio request blocks
         */
        audioIOB[i]->ioa_Request.io_Device = auDevice;
        aulockIOB[i]->ioa_Request.io_Device = auDevice;

        audioIOB[i]->ioa_Request.io_Message.mn_ReplyPort
                        = auReplyPort;
        aulockIOB[i]->ioa_Request.io_Message.mn_ReplyPort
                        = auLockPort;
}
for(i=0; i<4; i++)
{
        chan = GetAnyChannel(audioIOB[i]);

        printf("Got channel %ld\n",chan);

        /* Make the allocation keys match */

        aulockIOB[i]->ioa_AllocKey = audioIOB[i]->ioa_AllocKey;

        /* Make the unit numbers match */
        auunit = audioIOB[i]->ioa_Request.io_Unit;
```

Listing 8.7: The audio program (continued)

```
                aulockIOB[i]->ioa_Request.io_Unit = auunit;

                LockAChannel(aulockIOB[i],chan);

                /* If CheckIO returns true, it means the request has
                 * been returned.  This means the channel has been
                 * stolen.
                 */

                if(CheckIO(aulockIOB[i]))
                {
                        finishup("A channel was stolen!");
                }

        }
        for(i=0; i<4; i++)
        {
                /* now assuming nothing stolen, setup and request
                 * an output from that channel.
                 */
                audioIOB[i]->ioa_Data   = (UBYTE *)chipaudio;
                audioIOB[i]->ioa_Length = 8/2;  /* 4 WORDS in table */
                audioIOB[i]->ioa_Period = period[i];  /* from table */
                audioIOB[i]->ioa_Volume = 64;           /* maximum */
                audioIOB[i]->ioa_Cycles = 10000;       /* 10K times */

                audioIOB[i]->ioa_Request.io_Command = CMD_WRITE;
                audioIOB[i]->ioa_Request.io_Flags = ADIOF_PERVOL;

                 /* copy the audio block for freeing channels later */

                *aufreeIOB[i] = *audioIOB[i];

                printf("sending a request\n");
                BeginIO(audioIOB[i]);
        }
        for(i=0; i<4; i++)
        {
                WaitIO(audioIOB[i]);
        }
        for(i=0; i<4; i++)
        {
                FreeAChannel(aufreeIOB[i]);
                printf("freeing a channel\n");
        }
        finishup("Done!\n");
}


        finishup(string)
        char *string;
        {
                int i;
        if(auDevice)
           CloseDevice(audioIOB[0]);
        if(chipaudio)
           FreeMem(chipaudio,8);
        for(i=0; i<4; i++)
        {
                if(audioIOB[i])
                   FreeAudioIO(audioIOB[i]);
                if(aulockIOB[i])
                   FreeAudioIO(aulockIOB[i]);
```

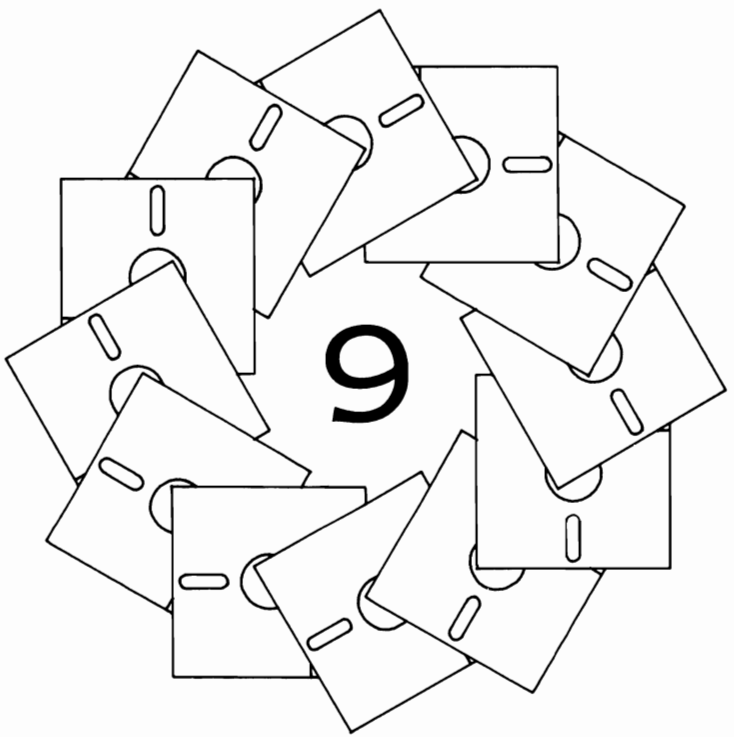
Listing 8.7: The audio program (continued)

```
        }
        if(auReplyPort)
           DeletePort(auReplyPort);
        if(auLockPort)
           DeletePort(auLockPort);

    printf("%ls\n",string);
    exit(0);
    }

    #include "ram:newaudio.c"
    #include "ram:lockchan.c"
    #include "ram:getaudio.c"
```

Listing 8.7: The audio program (continued)

If you want to learn more about digital sound synthesis, I recommend *Musical Applications of Microprocessors* by Hal Chamberlain (Hayden, 1986). In addition, a good treatment of both the sound and speech capabilities of the Amiga can be found in the book *Inside the Amiga* by John Thomas Berry (Indianapolis: Sams, 1986).

Multitasking



As introduced in Chapter 3, Exec has the ability to handle many tasks at the same time. This chapter provides some details about how Exec manages multitasking and provides examples that show how to create new tasks and how to run more than one task at a time.

TASKS

Exec shares the facilities of the processor by allowing the creation of tasks. Each task in the system has exclusive use of the machine registers while it is running its program code. If you create and start a task, then that task can share the facilities of the machine with all of the other tasks currently in the system.

A task is defined by a task control block. In this block are the parameters that define the operating state of the task, as well as storage areas in which the contents of the machine registers are kept when the task is temporarily suspended.

Every time a system interrupt occurs, Exec examines the current task and the list of tasks that are ready to run. Exec determines which task should currently be running by comparing the priority of the running task to the priority values for all of the other tasks that are ready to run. If another ready task has an equal or higher priority, then the complete state of the 68000 is saved in the task control block for the current task and the complete state of the new task is loaded from its own task control block and is started running, becoming the current task. Tasks of equal priority will continuously swap in and out of the processor, making it appear that all tasks are running at the same time.

There are some limitations on the functions a task can perform. For example, a task cannot do anything that requires any of the AmigaDOS functions. The design of AmigaDOS calls for additional variables to be associated with the task data structure. AmigaDOS uses these other variables to receive messages, or to keep track of where the program code and data was loaded for this program, or to keep track of the lock on the current directory, and so on.

Something to keep in mind is that only the task with the highest priority gets a chance to run. If you create a high priority task that does no I/O and performs no delay or wait functions, it is possible for your task to hog the machine entirely. If a higher priority task hogs the machine, any lower priority tasks may never get a chance to run at all.

Priorities of tasks can be set to any value from −128 to +127. Most of the time, tasks are started with a priority of zero. If all tasks have a priority of zero (as AmigaDOS uses when starting your program from the CLI), then each task has an equal chance to run and all tasks will

appear to be running simultaneously. All tasks of equal priority will round-robin; that is, as each is deactivated in favor of the next task on the task-ready list, the most recently deactivated task goes to the tail end of the list. As each ready task is taken from the head of the list, the task that is added to the end eventually reaches the head of the list and runs again.

PROCESSES

AmigaDOS handles multitasking by building its own structure called a process on top of the Exec task data structure. Most of the Process data structure is involved with the internals of AmigaDOS and, therefore, the items in the process control block are not described in this book. The Process data structure is listed in the *Amiga ROM Kernel Manual* in the Include file named libraries/dosextens.h.

Processes carry more overhead. That is, there are more things that AmigaDOS will do with processes than with tasks. However, other than specifying that tasks cannot perform AmigaDOS functions, this book will not go further into the distinction between processes and tasks. However, you will find examples of starting both processes and tasks from which you will be able to create your own applications.

THE EASY WAY TO START SOMETHING NEW

Before getting to the more complicated way of doing things (using tasks and processes) here is the simple way: you can use the AmigaDOS facilities for starting a separate process and automatically continue with your own program's operation while the other process runs right along with yours.

If you have used the CLI, you already know a way to start several programs running at the same time. For example, you can type

```
RUN clock
```

and

```
RUN notepad
```

This opens two new windows, each of which has the named program running. You still have the original CLI available in which you can type additional commands. Or you can type

```
NEWCLI
```

and get another CLI window from which other commands can be issued.

To easily start other programs from within one of your own programs, you can call Execute with a command string that is identical to what you would type directly into the CLI to start an independently running program. Just as you might type

```
RUN SOMETHING
```

into the CLI, you can add to your program a function call to Execute such as

```
success = Execute("RUN SOMETHING",0,0);
```

where SOMETHING is the name of the program you want to start running.

As you'll notice when you type the RUN command into the CLI, AmigaDOS returns control immediately to the CLI so that you can continue to type other commands. The same is true with the Execute function. RUN loads and begins to run SOMETHING, returning control to your program, with both programs running at the same time. You can start several programs running in this manner if you wish. The return value for success is always TRUE. That is, Execute always succeeds for a RUN command.

If you wish, you can also include redirection symbols in the command string so that the input and the output of this independent task could come from a disk file and go to a disk file. This would take the form

```
success = Execute("RUN < inputfile > outputfile SOMETHING",0,0);
```

where inputfile would be the name of the file from which input would be taken and outputfile would be the name of the file into which all output from SOMETHING would be placed as the program runs. The arrows in the command line are the redirection symbols of AmigaDOS. These are explained in Chapter 2.

Once the RUN command begins, AmigaDOS starts a separate process running for this other program. Both your process and this separate process are allowed to run, sharing the facilities of the machine.

This simple form of starting programs that run concurrently actually spawns (starts) an AmigaDOS process for each new running program. If you do not specify any redirection in the command string, then all output is directed to the CLI from which you started the original program. You need not worry about what kinds of functions your program uses because there is a complete process control block set up by AmigaDOS. So you can feel free to call AmigaDOS functions within the programs you spawn.

Again, if you are only setting up tasks, rather than processes, you may have to be more careful about which kinds of function calls you

include in those tasks. In other words, you will have to avoid using anything that directly or indirectly calls AmigaDOS. Directly calling AmigaDOS means directly using AmigaDOS functions described in Chapter 2 (such as Open, Read, Write, Close, and Delay). Indirectly calling AmigaDOS means using any of the I/O functions built into the Amiga C library.

I/O functions are defined as anything that moves data into or out of your program from any peripheral device. Among these functions are getchar, putchar, fopen, fclose, printf, and fprintf. The general guideline you can use to recognize an I/O function is to ask yourself: "Does the function move data through a peripheral such as the keyboard or screen or disk?" If so, it might not be possible to ask a task to do it. However, a process can do anything, so you might want to use a process instead of a task.

Why use a task if a process is at least as good? Well, sometimes you might want to minimize the space that your program uses, and if a task can do what you need to have done, then using a task might just be the logical choice.

Why get into tasking and creating processes when it is easy to start a separate program from the CLI or from another program? Well, sometimes you want to customize the way the system runs and perhaps have tasks communicating and cooperating with one another. The tasking example that follows shows how the task control block can be extended to provide intertask communications.

A TASKING EXAMPLE

The tasking example in this section includes a tool for initializing a task control block and starting a separate task. For convenience, both the main program and the task it starts are contained in the same block of code. This is a very simple program in which neither the main program nor the little task that it spawns do anything really significant. It is provided for illustration purposes only. The main program simply initializes the little task, then sends it a startup message, then goes to sleep waiting for the reply.

The main program could just as easily have been made to do other things while waiting for the little task to respond. For example, main could be reading data from the user, and the little task could be doing some calculations on previous data while main was getting new data. (Specifically, imagine the little task calculating the next move of a chess program while main was simply updating the move clock and waiting for the user to indicate the next move.)

Once the little task has responded to the startup message, it goes into a forever loop (by using Wait). Normally, when you write a program, such as

```
main()
{
   printf("Hello world\n");
}
```

it is perfectly OK to "fall off the end" of your program. In other words, you can omit any exit or return statement, since this is assumed by the C compiler. Once your program completes, it returns to the startup code (AStartup or LStartup) with which you linked it in the first place. That startup code takes care of initializing your program correctly as well as cleaning up after your program is finished.

If you launch a task, it is not possible to simply fall off the end, because the system will not have a special startup code associated with that task, and it will not know where to go and what to do. You have two possible choices for ending your task code. You can force your task into an endless wait. In this case, AmigaDOS stops the main program when it finishes, and main frees the memory for the task you spawned and deletes the task from the system task list. Or you can cause your task to delete itself as its final action. That method is shown here.

The Link File for the Task Example

The file in Listing 9.1 controls the linkage for the tasking example. To compile this program, use the steps on the following page.

```
; tasklink.lnk
;
; assign lib dfl:lib
; (dfl: is C development disk.)
; NOTE.... complile with the -v option in LC2 to disable stack checking
; code.... otherwise linker coughs with undefined items _cxovf and _base.
; (because the example does not use Lstartup.obj and lc.lib.)
;
; IF RUN FROM CLI:

FROM lib:Astartup.obj ram:tasking.o ram:inittask.o
TO ram:tasking
LIBRARY lib:amiga.lib
```

Listing 9.1: The tasking.lnk file

1. From your own disk, type

```
COPY tasking.c ram:
COPY inittask.c ram:
COPY tasklink.lnk ram:
```

2. With the Amiga C disk in drive 1, type:

```
   CD df1:examples
EXECUTE make ram:tasking.c
EXECUTE make ram:inittask.c
df1:c/alink with ram:tasklink.lnk
```

   Your result will be a program called tasking.

3. From the CLI, issue the command

```
RUN ram:tasking
```

   and you will see the results.

The Main and Little Task Program

Listing 9.2 is a fully commented listing that shows both the main program and the little task that it spawns.

The Initialize Task Function

Listing 9.3 contains the InitTask function used in Listing 9.2. It differs from the CreateTask function supplied in the Amiga function library in that the task that is created is not automatically started. The example tasking program waits for a startup message instead of starting immediately. Additionally, the main program allocates and deallocates memory for the task, whereas in CreateTask most of the memory allocation is done by CreateTask rather than by the process that spawns the task.

A PROCESSING EXAMPLE

If your code needs to perform I/O or to use AmigaDOS functions in any way, it will have to be spawned as a process rather than as a task. This still falls under the heading of multitasking, but it is done at a higher level, i.e., under control of AmigaDOS.

There are two programs contained in this section. The proctest program loads and starts littleproc, and unloads its code and data when it

```
/* tasking.c */

/* Sample tasking program -

        * Creation of a parent and child task.

        * Main task allocates space for a task control block and a
        * message port, to be dedicated to the child task.

        * Main initializes both port and tcb, then adds the child task.

        * Child goes to sleep on creation, waiting for a message
        * to be sent to its message port.

        * On sending that message, parent task goes to sleep waiting
        * childs reply.

        * When parent's message arrives on child's port, child
        * task awakes, retrieves the message and replies.  Goes into
        * an endless sleep (designer's choice).

        * Parent awakens on receipt of message at its reply port,
        * deallocates child memory utilization, and exits. */

/* system software version: V1.1 or higher  */

/* program dependency information:

        Link with Astartup.obj, InitTask.o, amiga.lib
        to become
                        tasking */


#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/tasks.h"
#include "exec/execbase.h"

#include "exec/io.h"

#define PRIORITY 0
#define STACKSIZE 500

extern struct Message   *GetMsg();
extern struct MsgPort   *CreatePort();
extern struct MsgPort   *FindPort();
extern APTR             AllocMem();
extern struct Task      *FindTask();
extern int              InitTask();

struct MyExtendedTask {
          struct Task          met_Task;       /* a task control block */
          struct MsgPort       met_MsgPort;    /* to a message port */
          int                  met_Status;     /* a status value for info */
};

littletask()
{
        int signalbit;  /* signal bit value for error checking */
```

Listing 9.2: The main and little task program

```
/* pointer to little task's message port */

struct MsgPort *mp;

/* pointer to little task's message */

struct Message *msg;

/* pointer to a task */

struct MyExtendedTask *met;

met = (struct MyExtendedTask *)FindTask(0);

/* use this to mean everything is OK so far */

met->met_Status = 0;

/* Point to the message port */

mp = &(met->met_MsgPort);

/* Now we have to set up the message port so that task can
 * go to sleep waiting for a message to arrive there.
 *
 * Tell the port which task gets signaled when this port
 * receives a message.  Nothing happens yet since mp_Flags
 * is set to PA_IGNORE by the master task. */

mp->mp_SigTask = (struct Task *)met;

/* Now get a signal bit number... this won't fail because
 * this task is brand new and has plenty of signal bits
 * to go around.  Error checking done anyway, but what to
 * do if it fails is up to you. */

signalbit = AllocSignal(-1);    /* allocate ANY signal bit */
if(signalbit != -1)
{
        /* A valid signal bit has been allocated! */
        /* NOW, tell the port to signal us if a message is received */
        mp->mp_Flags = PA_SIGNAL;
}
else
{
        /* no signal bit was available */
        met->met_Status = -1;
        goto finish;
}
/* If there was an error, then the flags variable never gets set
 * to other than PA_IGNORE.  As a result, this poor little task
 * might have to go to sleep forever waiting for a signal that will
 * never happen.  Thus, a master task/process could send a message
 * to this little task, then wait for a limited time, and finally
 * if no response from the little task, it can check the taskstatus
 * to see if the little task actually started.  If it did not,
 * then the error code or current status can be in the status
 * variable.  Instead of forcing this action, we have chosen
 * to terminate the task instead. (goto finish) */

WaitPort(mp);     /* wait for signal bit to be set */

/* If there is a message already there, task never even sleeps */
```

Listing 9.2: The main and little task program (continued)

```
        msg = GetMsg(mp);

        /* successfully responded to a message */

        met->met_Status = 1;

        /*      DO SOME OTHER GOOD STUFF HERE.... WHATEVER
                THE TASK WAS SUPPOSED TO DO */

    finish:

        Forbid();       /*  Disable task switching */
        ReplyMsg(msg);  /*  Send message back to starter */

        /* You would expect to free this signal bit using FreeSignal
         * in about this position.  However, since the task is about
         * to be removed anyway, why bother. */

        RemTask(0);     /* Permit() happens automatically when remove task */

        /* Remove task from task list; the next ready task
         * can begin to execute.  We can use this because of the
         * MemList stuff in InitTask().... any memory
         * on the task's memlist is returned to the system
         * automatically when RemTask is performed. */

        /* NOTICE that littletask MUST end in some kind of a
         * Wait() or an endless loop, or somehow cause itself
         * to be removed as is shown here.  If it falls off the end
         * nobody to return to.  It must hold firm while the
         * master task later deletes it. */
}               /* end of little task */

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


/* ****************************** NOTE ****************************

        This example does NOT have littletask try to printf anything
        because littletask is only a task, not a process.  When you
        RUN something (or simply start something from a CLI), that
        program gets attached to a process, which is a superset
        of a task.  Those functions described in the ROM Kernel
        Manual can be run by tasks.  Those functions described in
        the AmigaDOS Developers' Manual or the Lattice C manual
        must be called from main() or any of its own subroutines.
        Separate items, fired up as tasks rather than processes,
        must use only task-able functions.  (Delay(xx) is a DOS
        function, so littletask can't do that one either.)

        (printf implies an output to an AmigaDOS controlled CON: or
         RAW: window, is therefore an AmigaDOS interactive command
         and requires a process call it rather than a task)

        A user of tasks must be especially careful about controlling
        the things that are requested by a task, primarily things
        that are entirely self-contained code are probably the best
        to use.   Anything that may utilize the DOS should be avoided.
        This includes opening a library, a device or a font, since
        each of these causes a disk access (to load nonresident code
        or devices).   You may discover work-arounds, but the general
```

Listing 9.2: The main and little task program (continued)

```
        approach is that if your code has to do something that uses
        AmigaDOS, then you should probably spawn a PROCESS rather than
        a task.

        A separate example shows how to spawn a process rather than
        a task.

        This tasking example works for resident code,
        where both the main() and its tasks are loaded at once.
        The process example actually loads the other program as
        a separate item and unloads its code when it finishes.

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

main()
{
        struct Message mymessage;       /* an actual message data structure */
        struct MsgPort *mainmp;         /* pointer to main's reply port */
        struct MyExtendedTask *met;     /* pointer to an extended task */
                                        /* control block */
        struct MsgPort *mp;             /* pntr to littletask's message port */
        int result;                     /* nonzero if InitTask works OK */

        printf("\n Started main()\n");

        mainmp = CreatePort(0,0);

        /* Using for reply only, so don't we need to name it... */
        /* address for the reply is contained in the message itself. */

        if(mainmp == 0) exit(20);       /* error during createport */

        /* Set up the message data structure so that we can use PutMsg */
        /* to transmit this to the little task we are creating. */

        mymessage.mn_Node.ln_Type = NT_MESSAGE;
        mymessage.mn_Length       = sizeof(struct Message);
        mymessage.mn_ReplyPort    = mainmp;

        /* Now allocate space for an extended task control block and */
        /* initialize it */

        met = (struct MyExtendedTask *)AllocMem(sizeof
                        (struct MyExtendedTask), MEMF_PUBLIC | MEMF_CLEAR );

        if(met == 0)
        {
                /* Error during AllocMem */
                DeletePort(mainmp);
                exit(40);
        }

        /* Now initialize the task control block part of the extended task */

        result = InitTask( (struct Task *)met, "littletask",
                                PRIORITY, STACKSIZE );

        if (result == 0)
        {
                /* error during InitTask... no mem for stack */
                DeletePort(mainmp);
                exit(45);
        }
```

Listing 9.2: The main and little task program (continued)

```
        /* Get the address of the message port */

        mp = &(met->met_MsgPort);

        /* initialize the message port for the little task. */

        mp->mp_Node.ln_Type = NT_MSGPORT;       /* is a message port */
        mp->mp_Flags = PA_IGNORE;               /* when message arrives, */
                                                /* don't try to signal */
        NewList(&(mp->mp_MsgList));             /* initialize message list */
        /* Now that the message port is set up, it is legitimate to send a */
        /* message to it.  We can send a message before or after adding */
        /* the task */

        PutMsg(mp,&mymessage);

        AddTask( met, littletask, 0 );

        printf("\n Created and added the little task");

        WaitPort(mainmp);       /* wait for its reply */

        /* This example assumes that everything will go OK.  However
         * if there is an error, littletask will wait forever, and so
         * will main (since the message will never get back to the
         * reply port).  The alternative is to set up a timer, and call
         *
         *
         * wakeupmask = Wait( TIMER_SIGNAL_BIT | REPLYPORT_SIGNAL_BIT );
         *
         * then if no response after a reasonable time, examine the
         * met_Status to find out what went wrong with poor littletask
         * and do something about it. */

        GetMsg(mainmp); /* remove the message */
        printf("\n main: Little task received my message\n");

cleanup:
        if(met) {
          /* remove the littletask before we exit */
          FreeMem(met, sizeof(struct MyExtendedTask));
        }

        printf("\n main: Freed memory that littletask used");

        if(mainmp) {
          /* and our message reply port */
          DeletePort(mainmp);
        }

        /* Delay(250);          * Wait 5 seconds before exiting so
                                * that user can read the messages
                                * that we output into the Lattice
                                * Workbench window
        */

}       /* end of main */
```

Listing 9.2: The main and little task program (continued)

```
/* ********************************************************************** */
/* inittask.c -

        1.  Use memory that has been allocated by somebody else, and
            let them deallocate it later, as well.

        2.  Initialize as much of the task control block as is appropriate
            (same amount of init as CreateTask does).

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

/* From original code by Carl Sassenrath and Neil Katin; modified to let
 * main program do more initialization (extended task control block) before
 * actually firing off the task. */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/memory.h"
#include "exec/interrupts.h"
#include "exec/ports.h"
#include "exec/libraries.h"
#include "exec/tasks.h"
#include "exec/execbase.h"


/*  Initialize a task with given name, priority, and stack size. */
/*  It will use the default exception and trap handlers for now. */

/* The template for the mementries.  Unfortunately, this is hard to
 * do from C: mementries have unions, and they cannot be statically
 * initialized...
 *
 * In the interest of simplicity I recreate the mem entry structures
 * here with appropriate sizes.  We will copy this to a local
 * variable and set the stack size to what the user specified,
 * then attempt to actually allocate the memory. */

#define ME_STACK        0
#define NUMENTRIES      1

struct FakeMemEntry {
    ULONG fme_Reqs;
    ULONG fme_Length;
};

struct FakeMemList {
    struct Node fml_Node;
    UWORD       fml_NumEntries;
    struct FakeMemEntry fml_ME[NUMENTRIES];
} TaskMemTemplate = {
    { 0 },                                          /* Node */
    NUMENTRIES,                                     /* num entries */

    {                                               /* actual entries: */
        { MEMF_CLEAR,   0 }                         /* stack */
    }
};

int
InitTask( task, name, pri, stackSize )
    struct Task *task;
    char *name;
    UBYTE pri;
```

Listing 9.3: The inittask function

```
    ULONG stackSize;
{
    struct Task *newTask;
    struct FakeMemList fakememlist;
    struct MemList *ml;

    /* round the stack up to longwords... */
    stackSize = (stackSize +3) & ~3;

    /* This will allocate one chunk of memory: */
    /* a stack of PRIVATE */
    fakememlist = TaskMemTemplate;

    fakememlist.fml_ME[ME_STACK].fme_Length = stackSize;

    ml = (struct MemList *) AllocEntry( &fakememlist );

    if( ! ml ) {
        return( 0 );
    }

    /* Set the stack accounting stuff */
    newTask = task;

    newTask->tc_SPLower = ml->ml_ME[ME_STACK].me_Addr;
    newTask->tc_SPUpper = (APTR)((ULONG)(newTask->tc_SPLower) + stackSize);
    newTask->tc_SPReg = newTask->tc_SPUpper;

    /* Misc task data structures */
    newTask->tc_Node.ln_Type = NT_TASK;
    newTask->tc_Node.ln_Pri = pri;
    newTask->tc_Node.ln_Name = name;

    /* Add it to the tasks memory list */
    NewList( &newTask->tc_MemEntry );
    AddHead( &newTask->tc_MemEntry, ml );

    return( 1 );
}
```

Listing 9.3: The inittask function (continued)

finishes. Thus, littleproc is spawned by proctest. The startup code with which it is linked automatically waits for a Workbench startup message before it gets going. Using the same message port that was provided when the process was initiated, it again goes to sleep waiting for a message that contains specific information—in this case, the parameters that the master program is using, namely its stdout and stderr file handles. Thus, this spawned process can be made to output to the same window from which the originating process was begun.

A process is a superset of a task, and the various AmigaDOS routines require that a process control block and its associated information be available in order to run. This code is provided to allow a programmer who requires a process rather than a task to have an example on which to build.

The Link Files for the Process Example

To try this example, you will have to compile both programs separately. The link files (for use with the alink program) are provided here. The separate steps are as follows:

1. From your source file disk, type:

```
COPY process
COPY littleproc.c ram:
```

2. With the Amiga C disk in drive 1, type:

```
CD df1:examples
```

3. Use your favorite text editor to modify the file named make in the examples directory. Change the line that begins "lc2" to begin as "lc2 -v". This disables stack checking for the example.

4. Type the following lines:

```
EXECUTE make ram:process
EXECUTE make ram:littleproc

df1:c/alink with ram:process.with
df1:c/alink with ram:littleproc.with
```

This is the content of the file named process.with, which is the first link file:

```
FROM lib:Astartup.obj process.o
TO process
LIBRARY lib:amiga.lib
```

This is the content of the file named littleproc.with, which is the second link file:

```
FROM lib:Astartup.obj littleproc.o
TO littleproc
LIBRARY lib:amiga.lib
```

The Process Programs

To run Listings 9.4 and 9.5, make sure that they are both in the same directory because proctest will look in the same directory for the littleproc code. Then run proctest. It loads and executes littleproc, then unloads littleproc and exits.

```
/* proctest.c */

/* system software version: V1.1 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"
#include "exec/memory.h"
#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"

#define PRIORITY 0
#define STACKSIZE 5000

extern struct Message *GetMsg();
extern int LoadSeg();
extern struct MsgPort *CreateProc();
extern struct MsgPort *CreatePort();

struct MyMess {
   struct Message mm_Message;
   int            mm_OutPointer;
   int            mm_ErrPointer;
};

extern int stdout;
extern int stderr;

main()
{
        struct Message *reply;
        struct Process *myprocess;

        /* Message that we send to the process to wake it up */

        struct WBStartup *msg;

        /* Message to contain my own parameters to pass on to spawned
         * process, sample only.  Just to prove that we correctly
         * create a process, we are giving it something other than nil:
         * as its stdout and stderr... in fact, giving it OUR values
         * so we can share the same output window. */

        struct MyMess *parms;

        /* Because main() is itself started as a process, it automatically
         * has a message port allocated for itself.  Located at
         *  &((struct Process *)FindTask(0))->pr_MsgPort */

        int littleSeg;

        /* Actually littleSeg is a BPTR, but the int declaration
         * keeps the compiler happy and we don't use the
         * value ourselves anyhow... just pass it on. */

        char *startname, *parmname;
```

Listing 9.4: The proctest program

```
        struct MsgPort *mainmp;    /* pointer to main's msg port */
        struct MsgPort *littleProc; /* pointer to spawned proc's msg port */

/* Provide names for the messages we are passing so we can check the returned */
/* messages at the message ports.... that is if we choose to do so. */
        startname = "startermessage";
        parmname  = "parameterpass";

/* LOAD THE PROGRAM TO BE STARTED FROM MAIN ***************************** */

        littleSeg = LoadSeg("littleproc");
        if(littleSeg == 0)
        {
                printf("\nlittleproc not found");
                exit(999);
        }

/* CREATE A PROCESS FOR THIS OTHER PROGRAM ****************************** */

        littleProc = CreateProc("littleguy",PRIORITY, littleSeg, STACKSIZE);
        if( littleProc == 0 )
        {
                printf("\Couldn't create the process");
                UnLoadSeg(littleSeg);
                exit( 1000 );
        }

/* ********************************************************************** */
/* Locate the message port that is allocated as part of the process */
/* that started main() in the first place */

        myprocess = (struct Process *)FindTask(0);

        mainmp = CreatePort(0,0;

/* ********************************************************************** */
/* THE FOLLOWING CODE BLOCK STARTS THE PROCESS RUNNING,
                                AS THOUGH CALLED FROM WORKBENCH */

/*      In fact, because we created the process the way that is shown
        here, if you use the standard startup code, the program must
        be started as though called from Workbench.  It is now waiting for
        a startup message.

        (There is, in fact, another way to call a loaded program's code,
        but it does not entail starting another process.  Rather it
        uses a direct call (as a subroutine) to the loaded code.  The
        other program runs on your own stack, so your program must
        have sufficient stack to handle both.  It also runs
        under your own process, so your own program does not get
        control until that other program has completed.  The program
        return()'s or exit()'s to you, providing the appropriate
        returncode.)

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

        /* This message block is a wakeup call to the process we created. */
        msg = (struct WBStartup *)AllocMem(sizeof(struct WBStartup),
                MEMF_CLEAR);
        if(msg)
        {
                /* Preset the necessary arguments for message passing */

                msg->sm_Message.mn_ReplyPort = mainmp;
```

Listing 9.4: The proctest program (continued)

```
        msg->sm_Message.mn_Length = sizeof(struct WBStartup);
        msg->sm_Message.mn_Node.ln_Name = startname;

        /* Passing no workbench arguments to this process;
         * we are not WBench.  Of course, if we want to pass
         * workbench-style arguments this way, we can. */

        msg->sm_ArgList = NULL;

        /* If the process is being opened without a ToolWindow
         * (Workbench sets this up) as a parent, slave will simply
         * go on to do its own main()  ... as shown in Astartup.asm */

        msg->sm_ToolWindow = NULL;

        /* Send the startup message */

        PutMsg(littleProc,msg);
}
else
{
        printf("\nCouldnt allocate mem for WBStartup!\n");
        goto aarrgghh;  /* Oh no, a "goto" ! */
}
/* ********************************************************** */
/* Just a sample message, still using the same message and
 * reply ports
 *
 * Littleproc is a cooperating process...it KNOWS it must wait
 * until a message arrives at its port, containing the parameters
 * it should use for output.
 *
 * The startup message is handled by the standard startup code.
 * This parameter message is handled by the program code itself.
 * The startup message is returned to the replyport by the startup
 * code, after the program code exits or returns. */

parms = (struct MyMess *)AllocMem(sizeof(struct MyMess),MEMF_CLEAR);
if(parms)
{
        parms->mm_Message.mn_ReplyPort = mainmp;
        parms->mm_Message.mn_Length = sizeof(struct mymess);
        parms->mm_Message.mn_Node.ln_Name = parmname;

        /* NOTE THAT THESE ARE THE AStartup.asm stdout and stderr;
         * the example works only if both master and slave are
         * compiled and linked with the same startup code. */

        parms->mm_OutPointer = (int)stdout;
        parms->mm_ErrPointer = (int)stderr;
        /* send it our parameters */

        PutMsg(littleProc,parms);

        /* wait for the reply from parameter pass. */

        WaitPort(mainmp);

        reply = GetMsg(mainmp);

        /* Message node name should contain the address of the
         * string "parms" if error checking was included.
         *
         *
```

Listing 9.4: The proctest program (continued)

```
                * You should probably allocate separate ports for
                * parameter passing different from the main port
                * automatically allocated by the system when a
                * process is initiated.  It would alleviate
                * some of the checking that is appropriate to do
                * when multiple kinds of messages arrive at the same port.
                *
                *
                *       NOW MAIN CAN GO ON AND DO SOMETHING USEFUL,
                *       LATER CAN COME BACK AND SEE IF SPAWNED PROCESS
                *       HAS COMPLETED AND IS READY TO BE UNLOADED.
                *
                *
                * Wait for the return of the wbstartup message before
                * main itself is allowed to exit. */

               WaitPort(mainmp);

               reply = GetMsg(mainmp);
               /* Message node name should be
                * address of "startermessage"  */

               /* NOTE: there should be checking here to see if the message
                * received at this port was the string, or the wakeup call.
                * This sample code only assumes that the string is received
                * and replied first, then the wakeup call message is returned
                * as the little task is exiting. */

               UnLoadSeg(littleSeg);
               printf("\nSlave exited; Master unloaded its code and data\n");
        }
        else
        {
               printf("\nCouldn't allocate memory for parameter message\n");
        }
    aarrgghh:
               /* arrive here on good or bad exit */

        if(mainmp){ DeletePort(mainmp);
        if(parms) { FreeMem( parms, sizeof(struct MyMess));     }
        if(msg)   { FreeMem( msg,   sizeof(struct WBStartup)); }

}       /* end of main */
```

Listing 9.4: The proctest program (continued)

Note that these programs were written to use with the Amiga startup code (AStartup.obj) rather than the Lattice startup code (Lstartup.obj). No attempt has been made to adapt them to the Lattice code. The only point of incompatibility should be in the use of stdout and stderr. Lattice defines them differently. Lattice defines stdout and stderr as the address of an I/O block. If the main process is compiled under Lattice and the subprocess is also compiled under Lattice, then the values passed for stdout and stderr will be compatible.

AmigaDOS stdout and stderr are BCPL language pointers to an AmigaDOS data structure. When using amiga.lib to provide the printf function at link time, the amiga.lib version of printf internally uses the AmigaDOS

```
/* littleproc.c */

/* Sample slave code for create process test */

/* system software version: V1.1 */

#include "exec/types.h"
#include "exec/nodes.h"
#include "exec/lists.h"
#include "exec/libraries.h"
#include "exec/ports.h"
#include "exec/interrupts.h"
#include "exec/io.h"

#include "libraries/dos.h"
#include "libraries/dosextens.h"

#include "workbench/startup.h"

/* these are going to be supplied to the slave by the starter */
/* they are actually defined in the startup code (Astartup.asm) */

extern int stdout;
extern int stderr;

struct MyMess {
   struct Message mm_Message;
   int mm_OutPointer;
   int mm_ErrPointer;
};

extern struct Message *GetMsg();
extern struct Task *FindTask();
extern struct FileHandle *Open();

main()
{
        struct MyMess *msg;
        struct MsgPort *myport;
        struct Process *myprocess;

        struct FileHandle *myOwnOutput;

        myprocess = (struct Process *)FindTask(0);

        myport = &myprocess->pr_MsgPort;

        /* Wait for starter to post a message.  Special sample message
         * has its stderr, stdout so we can both post stuff to the
         * same CLI window as it started from */

        WaitPort(myport);
        msg = (struct MyMess *)GetMsg(myport);
        stdout = msg->mm_OutPointer;

        /* Use printf to prove that it is really a process...
         * a simple task cannot do this without crashing! */

        printf("\nHere I am, that slave process you started!!!");
        printf("\nNow going to open MY OWN window.\n");

        /* NOW DO SOMETHING USEFUL... DO WHATEVER THE PROCESS WAS DESIGNED
         * TO ACCOMPLISH. */
```

Listing 9.5: The littleproc program

```
        myOwnOutput = Open("CON:10/10/320/150/SlaveProcess",MODE_NEWFILE);
        if(myOwnOutput == 0)
        {
                ReplyMsg(msg);  /* tell main I'm done */
                exit(0);        /* can't return an error code anyhow */
        }
        else
        {
                stdout = (int)myOwnOutput;
                /* reset my output file handle */
                printf("See, I can do AmigaDOS!");
                Delay(250);     /* 250/50 = 5 seconds */
                stdout = msg->mm_OutPointer;
                Close(myOwnOutput);
                ReplyMsg(msg);
        }

        /* Now simply fall off the end of the world,
         * returns to the startup code, and should exit cleanly */

}
```

Listing 9.5: The littleproc program (continued)

version of stdout. If you link with lc.lib specified first, then that link file uses its own interpretation of stdout, which will not be compatible with the values received from the Open function (*Open* is AmigaDOS, *open* is Lattice).

Also note that the example utilizes the message port that AmigaDOS reserves for a process to use to receive AmigaDOS I/O messages. If you find this code interesting and useful, you should probably define and initialize a separate message port for your own messages rather than sharing the message port with AmigaDOS. Stealing the AmigaDOS message port is simply a convenient way to get things going. If your process needs to do extensive I/O, it is advisable to create a separate message port for your own use.

As a final note about this processing example, you could have created and started another process by using the Execute command of AmigaDOS:

```
success = Execute("someprogram",0,0);
```

But this example was provided to demonstrate interprocess communication setup and message passing.

INTERTASK COMMUNICATIONS

For intertask and interprocess communications, the programs shown have used message ports that have been explicitly created. If you have

processes that are started entirely independently, there are several ways that you can have a process find out if another process or task is already running so that you can communicate with it or use something it has already created.

Finding Tasks

The system maintains many different lists, including a list of tasks and a list of ports. When you start a task running by using the CreateTask function, the function places the name of the task within the list node of the task control block. Thereafter, you can use the FindTask function to locate it.

In Listing 9.3, the InitTask function was given a name by which the task control block could be found. That name was littleguy. Instead of keeping track of the address of the task control block that the main program created, main could have added the task to the system, and found the task (and thereby found the message port) as follows:

```
struct Task *taskblock; /* a pointer to a task control block */

taskblock = FindTask("littleguy");
```

Thus, anything that you manage to associate directly with a task control block can be found after that task is added to the system task list. However, in Listing 9.3, since main created the block, the program knew where the block was.

Finding Processes

Unfortunately, finding processes is not as easy as finding tasks. Although AmigaDOS (as of release 1.1) uses the task list to keep track of processes, AmigaDOS does not initialize the task name field to relate to the name of the process that it is running. So it is not possible to identify a running process by examining the names on the task list.

Thus, if your program needs to identify a running program so it can pass data to another program, it is more appropriate to use a task than a process.

Finding Ports

If you name and create a message port and add it to the system list, your task or another task or process can find that port later by using the FindPort function:

```
myport = FindPort("thisport");
```

If the function returns a value of NULL, then there was no port by that

name in the system port list. If the value returned is not NULL, then myport points to the list node of the message port of the name that it found on the system port list.

I needed this capability for a graphics demonstration program where each of six independent programs required a four-bitplane-deep custom screen to demonstrate one of the many graphics functions available on the Amiga. One of these programs is called Rectangles; it creates multicolored rectangles in a graphics window. Another is called Lines; it draws random, colored lines. A third is called Wallpaper; it creates multicolored rectangles.

Custom screens can use a lot of memory, so I decided to make each program look to see if another program of its type was already running. If another program had already created a custom screen, each of my programs, instead of opening its own custom screen, would open a window on that other program's custom screen. Each window has its own close gadget, which closes that program down. Each window has its own title bar, which lists the names of all of the other programs that it knows uses that same custom screen.

To communicate between the running programs, I created a custom version of a message port:

```
typedef struct {
    struct MsgPort normalMsgPort;
    int usersOfScreen;
    struct Screen *screen;
    } MYCUSTOMPORT;
```

By initializing the normalMsgPort properly, it could be added to the system port list by using the AddPort function. Remember that the system does not care how large each list item is, as long as the List structure elements are initialized correctly.

The first program (whichever of the group of compatible graphics programs ran first) would allocate memory for this custom port, open a custom screen, copy the custom screen address into the custom port data structure, and initialize the usersOfScreen to 1. It could then add this port to the system port list and open a window on this custom screen.

Subsequent programs could find that port, using FindPort, and from the usersOfScreen value, calculate where to put a new window so that it would not totally obscure any graphics already running on that custom screen or windows opening on that screen. Each new program to find the port and open a window would increment the count of usersOfScreen.

Each time a program closed its window, the count of usersOfScreen would decrement. When the count got to zero, whichever program causes the count to get there deleted the screen, deleted the port, and ended its operation.

Listing 9.6 shows two code segments that are part of all of the graphics programs mentioned above. These segments demonstrate how the custom port serves as the rendezvous point for all programs once they are running in the system, seeking a custom screen on which to open.

```
/* code fragment number 1 - custom port */

/* Note: not all declarations are incorporated here.  This code is
 * just provided to give the potential user of multitasked applications
 * a few ideas. */

struct MyPort {                 /* custom constructed message port */
        struct MsgPort mp;      /* a standard message port */
        struct Screen *Screen;  /* where is the screen that is just mine */
        int Users;              /* how many programs are using this screen */
        int RangeSeed;          /* what is the current value of the random
                                 * number generator for the most recent
                                 * user... else RangeSeed starts at zero
                                 * for EVERYBODY */
        char portname[64];
        char screenname[64];
};

struct Screen *
Setup()                 /* Returns a pointer to a new or existing screen */
{
    int junk;
    struct MyPort *myp, *sysmyp;
    struct Screen *screen;
    GfxBase = OpenLibrary("graphics.library", 0);
    if (GfxBase == NULL)
    {
        problem = 1001;         /* Can't open gfx library */
        return(0);
    }
    IntuitionBase = OpenLibrary("intuition.library", 0);
    if (IntuitionBase == NULL)
    {
        problem = 1002;         /* Can't open intuition library */
        CloseLibrary(GfxBase);
        return(0);
    }


    /* Does another application already have a custom screen open?
     * If so, use it.  If not, open one.  Prevent multitask switching
     * here in case two of these are started simultaneously.  Only one
     * at a time should be allowed to see if the screen-info-carrying
     * port exists and then create it if it does not. */

    /* Begin to create a custom message port, just in case the system
     * doesn't already have one.   This way, we'll be ready to add it
     * and won't have to Disable() for very long at all */
```

Listing 9.6: Custom port code fragments

```
    myp = (struct MyPort *)AllocMem(sizeof(struct MyPort),MEMF_CLEAR);
    if (myp == 0)
    {
        CloseLibrary(IntuitionBase);
        CloseLibrary(GfxBase);
        return(0);
    }
    else
    {
        /* Setup the port parameters */

        myp->mp.mp_Node.ln_Pri = 0;
        myp->mp.mp_Node.ln_Type = NT_MSGPORT;
        NewList(&(myp->mp.mp_MsgList));

        /* Establish the title for the port itself, within the port */

        strcpy( &(myp->portname[0]) , "lowres.16.color" );
        myp->mp.mp_Node.ln_Name = &(myp->portname[0]);

        /* Establish the title for the test screen, within the port */

        strcpy( &(myp->screenname[0]), "TestScreen" );
        ns.DefaultTitle = (UBYTE *)&(myp->screenname[0]);

        /* Number of Users of a custom screen */

        myp->Users = 1;         /* One user so far; just opened it */

        /* Calculate a dummy value, to change current RangeSeed value */

        junk = RangeRand(100);

        /* Starting value for random number generator for next user */

        myp->RangeSeed = RangeSeed;

/* **** Start interrupt disabled Section ***************************** */

        Disable();      /* prevent task switching during this operation */

        /* We are ready to add our custom port to the system.  Is there
         * one like it already there?  If so, deallocate ours.  If not,
         * open a custom screen, and finish initializing our port with
         * its address.   Then add it to the system port list. */

        sysmyp = (struct MyPort *)FindPort("lowres.16.color");
        if(sysmyp == 0)
        {
            screen = OpenScreen(&ns);
            if (screen == NULL)
            {
                problem = 1003;         /* Can't open a custom screen */
                Enable();       /* Enable interrupts and task switching */
                FreeMem(myp, sizeof(struct MyPort));
                CloseLibrary(IntuitionBase);
                CloseLibrary(GfxBase);
                return(0);
            }

            /* Show where to find the custom screen that was just opened. */

            myp->Screen = screen;
```

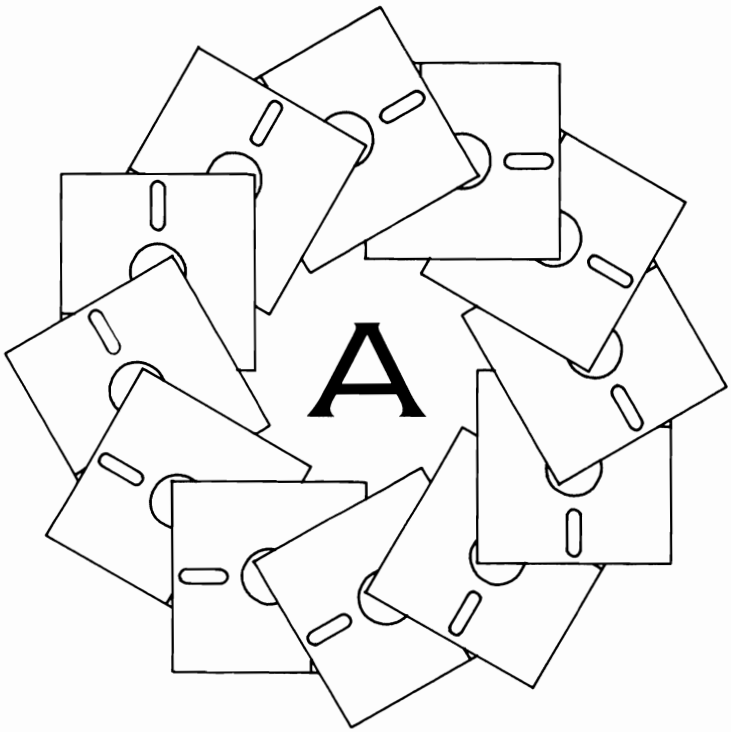
Listing 9.6: Custom port code fragments (continued)

```
            /* Add to system so others can find it */

            AddPort(myp);
            Enable();           /* Enable interrupts and task switching */
            return(screen);
        }
        else            /* The system already HAS this custom port! */
        {
            /* If system has a port like this, it doesn't need ours */

            FreeMem(myp, sizeof(struct MyPort));

            /* Add one to the number of users */
            sysmyp->Users += 1;

            /* Take prior user's RangeSeed value */
            RangeSeed = sysmyp->RangeSeed;
            junk = RangeRand(100);      /* Changes RangeSeed value */

            /* Save a new value for next user */
            sysmyp->RangeSeed = RangeSeed;
            Enable();                           /* Enable task switching */

            /* And return the screen location */
            return(sysmyp->Screen);
        }
    }
}

/* code fragment number 2 */

/* Close the custom window and decrement the number of users of the
 * custom screen.  If Users value drops to zero, also delete the screen
 * message port, free its memory and close the custom screen. */

int
EndTest()
{
    struct MyPort *myp;
    if (w != NULL) {
        ClearMenuStrip(w);
        CloseWindow(w);
    }
    Forbid();                           /* Momentarily halt task switching */
    myp = (struct MyPort *)FindPort("lowres.16.color");
    if(myp)                             /* This should succeed */
    {
        if((myp->Users -= 1)==0)
        {
            if (myp->Screen != NULL) CloseScreen(myp->Screen);
            RemPort(myp);
            FreeMem(myp, sizeof(struct MyPort));
        }
    }
    Permit();                           /* Enable task switching */
    return(TRUE);
}
```

Listing 9.6: Custom port code fragments (continued)

The multitasking capabilities of the Amiga offer significant opportunities to the developer. The code shown here has just barely scratched the surface. I hope, though, that this chapter has provided some insight into how the multitasking system works and a jumping-off point for future code development.

The Text Editor (ED)



The ED program is a simple text editor that you can use to create source files for your C compiler. For those of you who are unfamiliar with using a text editor to create a program, this appendix guides you through creating a short program.

As ED starts, it presents you with a blank screen on which you can type lines of text. After you have completed typing those lines, you can save this text to a file.

ED is a line editor, not a word processor. As you use ED, keep in mind that although it presents you with a full screen of text with which to work, it treats each line as an individual entity. For example, a block of text must consist of one or more lines. A text block mark cannot be placed in the middle of a line. Likewise, when you move or copy a block of text, ED inserts the block in between the line in which the cursor currently resides and the line immediately above it.

As you gain experience with ED, you may tend to use more of its commands. However, a beginning user need only remember a few basic rules and commands in order to use the program effectively. Here are some points to remember:

- You are always in insert mode. Wherever the cursor is located, if you type a legal (noncommand) character, ED will insert that character at the cursor position and push anything else to the right. A Return key press in the middle of a line splits the line.
- The cursor keys work as expected. You can move through the file exclusively with the cursor keys.
- The Backspace key deletes the character to the immediate left of the cursor.
- The Esc key is used for extended commands. If you press the Esc key, the cursor temporarily moves down to the status line at the bottom of the screen and waits for you to complete the extended command.

To start ED, make sure your Workbench or CLI disk is in the internal drive. This disk must be write-enabled to allow the ED program to create a work file in the SYS:T directory. From a CLI window, type

```
ED hello.c
```

and press Return. A new window opens and ED displays:

```
Creating a new file
```

Your cursor is at the top of the window. Type the following lines exactly as shown, pressing the Return key at the end of each line:

```
main()
{
   printf("\nHello world\n");
}
```

Then press Esc X and Return. Now your program is saved so that it can be used later by the compiler.

Table A.1 summarizes the ED commands. In the table, the ^ notation indicates a control command. This means that you must hold down the Ctrl key then press the specified command letter to execute a particular command. Although the command letters appear in uppercase, you can use lowercase letters and get the same result.

Cursor Movement Commands

↑	Move one line up.
↓	Move one line down.
→	Move one character to the right.
←	Move one character to the left.
^I	(Tab) Move to next tab stop.
^R	Move to end of previous word.
^T	Move to start of next word.
^D	Scroll text down.
^U	Scroll text up.
^E	Move to top or bottom of screen.
^]	Move to start or end of line.
^M	(Return) Move down one line and to the left margin.
Esc B	Move to end of file.

Table A.1: ED commands

Esc T	Move to start of file.
Esc N	Move to start of next line.
Esc P	Move to start of previous line.
Esc CE	Move to end of current line.
Esc CB	Move to start of current line.
Esc M <line-number>	Move to a specific line number within the file.

Insert and Delete Commands

^A	Insert a line after the current line.
^B	Delete the line in which the cursor is located.
^H	(Backspace) Delete the character to the left of the cursor and move everything back one space.
Del	Delete the character on which the cursor is sitting.
^O	If the cursor is on a space character, delete all spaces up to the next word on the line. If the cursor is on a nonspace character, delete this and all characters to the right until the next space character is encountered.
^Y	Delete to the end of the line, including the character on which the cursor is resting.
Esc A /<string>/	Insert this string of characters as a line preceding the current line.
Esc I /<string>/	Insert this string of characters as a line following the current line.
Esc D	Delete the current line.
Esc DC	Delete the character at the cursor.

Table A.1: ED commands (continued)

Esc IF !<pathname>!	Insert the file with this name at the current cursor position.

Margin and Tab Commands

Esc SL <number>	Set the left margin at the column number specified. (The default is 1.)
Esc SR <number>	Set the right margin at the column number specified. (The default is 80. The maximum is 255, since this is the maximum line length the ED program and AmigaDOS allow.)
Esc ST <number>	Set the tab distance to this number. Position the standard tab stops this far apart.
Esc EX	Extend the right margin. (Same as margin release on a typewriter.)

Find and Replace Commands

Esc F /<string>/	Search forward for the next occurrence of the specified string of characters.
Esc BF /<string>/	Search backward for an occurrence of the specified string of characters.
Esc E/<oldstring> /<newstring>/	Locate the next occurrence (forward search) of the specified old string and replace it with the new string. Do not verify, just replace without asking.
Esc EQ/<oldstring> /<newstring>/	Locate the next occurrence (forward search) of the specified old string and ask for verification—if it is OK to replace this particular occurrence.
Esc LC	Treat upper- and lowercase characters as different when performing searches.

Table A.1: ED commands (continued)

Block Commands

Esc BS	Mark this line as the beginning of a block.
Esc BE	Mark this line as the end of a block.
Esc DB	Delete this entire block.
Esc IB	Copy the marked block to the current cursor location.
Esc SB	Show the top line of the marked block as the top line of the screen. (Lets you quickly move to a marked position within the file.)
ESC WB !<pathname>!	Write the marked block out to a specified file. If the path name is not simply to the current directory, you can specify the complete path name, including slashes. (The exclamation points are used to delimit the path name.)

Save and Quit Commands

Esc SA	Save the file to the current file name and continue editing.
Esc Q	Quit without saving any changes made since most recent save. ED asks you to verify that it is OK to quit.
Esc X	Exit the program, saving the changes to the current file name.

Miscellaneous Commands

Esc J	Combine this line and the next line as a single line.
Esc S	Split the current line at the cursor. Take the character at the cursor position and make it the first character on the following line.
Esc SH	Show the status of the editor.
Esc V	Redraw the screen.

Table A.1: ED commands (continued)

Note that both Esc SA and Esc X accept an optional file name to be used in place of the file name used to open the file. This takes the form

```
Esc X !<pathname>!
```

or

```
Esc SA !<pathname>!
```

These commands can be useful for saving intermediate forms of the editing you are performing.

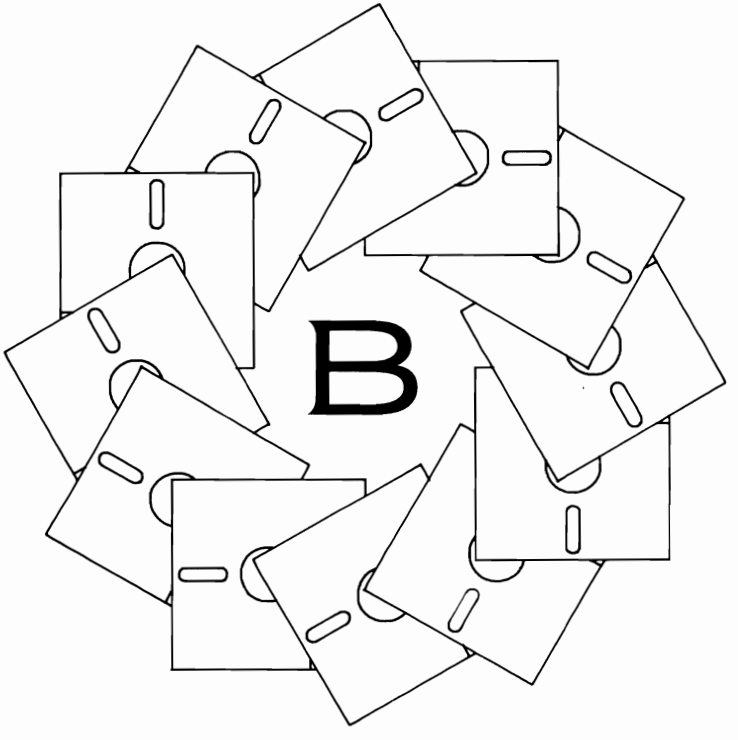
Once you have pressed Esc, your command line can contain multiple commands, separated by a semicolon. For example

```
F /a certain phrase/ ; E/oldword/newword/
```

searches for the string "a certain phrase", then following that phrase, searches for the next occurrence of "oldword" and substitutes "newword" for it.

You can specify how many times a command (or a command line) should be repeated by placing a number ahead of the command to be repeated. Alternatively, in place of the number, you can specify RP, which repeats the command until the end of the file.

The Amiga C Compiler



This appendix explains how to run the Amiga C compiler, which is a derivative of Lattice C. Readers who are already familiar with the Lattice C compiler on other systems should have little difficulty adjusting to this version of the compiler.

I have tried to keep compiler-specific details out of the book wherever possible. However, to ensure that there is a common basis for discussion, I used the Amiga C compiler for all of the examples. If you are using a different compiler, it may be necessary for you to modify the programs to adapt to your own compiler. The examples have been kept as short and direct as possible so as to minimize any such adaptation.

RUNNING THE AMIGA C COMPILER

Once you have created your program—using Ed, MicroEmacs, or some other program text editor—there are three separate steps you must perform before your program is ready to run.

Compiler First Phase

The first phase is called LC1. In this phase, your program is translated from C into an intermediate compiler code. During this phase, program syntax and structure are checked. Typically, you run this first phase by typing this command line:

```
CD DF1:C ;change directory to Amiga C compiler disk
LC1 -i:include/ df0:hello.c to df0:hello.q
```

(I am assuming that you are working with a 512K Amiga with an external disk drive containing your Amiga C compiler disk and that the program to compile is located on a disk in the internal drive and is named hello.c.)

This command sequence produces a file named hello.q on the internal disk drive in the root directory. The phrase

```
-i:include/
```

tells the compiler which directory it is to search to locate any files you may have requested to be included for the compilation, where the colon means the root of this disk (df1:) and include/ is the prefix for searching for any Include file you have specified in your program.

The output file will be named hello.q automatically (unless you provide a different name). I specified it here so you could see what the compiler is expected to output. The q extension stands for quad-file, the name that Amiga C uses for intermediate output from the first phase of the compilation.

Compiler Second Phase

The second phase is called LC2. During this phase, the intermediate language that the first phase produced is translated into an object file. Assuming that you have successfully completed phase one, you execute phase two by typing the command line

```
LC2 df0:hello.q to df0:hello.o
```

This produces the object file for your program.

Compiler Third Phase

The object file is not the final version of your program. It contains all of the translated instructions for what you wish to do, but does not, at this point, contain the code for any system routines you may have used.

To make a complete working program, you must perform a final phase, called ALINK that combines your program with selected object code from one or more function libraries (debug.lib or amiga.lib) and one or more previously compiled object files.

Working with a high level language, such as C, is an advantage here. The result of your program compilation is a relocatable code file that AmigaDOS can load anywhere in the system memory and run as a process with other tasks in the system. Thus, whatever program you produce will be able to share the resources of the computer with other cooperating tasks.

If you have successfully performed phase one and phase two, here is a typical call to ALINK that turns hello into a working program:

```
ALINK FROM df0:hello.o + Lstartup.obj TO df0:hello
  LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
```

(Type all of this on the same command line.) If successful, this three-phase sequence results in a program that you can execute simply by typing:

```
cd df0:
hello
```

Summary of Compiler Calls

Here is a summary of the steps required to compile a program under Amiga C:

1. Create the program using your favorite text editor.
2. Run phase one of the compiler (LC1) producing a file whose name ends in .q.

3. Run phase two of the compiler on the .q file resulting in a file whose name ends in .o.

4. Run phase three of the compiler (ALINK), linking your program with the system object files and libraries.

The result is a working program.

CREATING AND USING A MAKE FILE

Running three different compile phases just to test a program may seem like a lot of typing. The Amiga C disk includes a command script file (called an execute file) that enables you to dispense with the three-phase compilation. This particular execute file is called a *make file* because it can be used to create or direct the creation of other files. The name of the file that will perform all three phases of the compilation is make, and it is located in the examples directory on the C disk. (On some early compiler disks, the file is called makesimple. On those same early disks, the file named make only performed the first two phases of compilation. A file named link performed the third phase. If you are producing a large program consisting of several C files, you might consider compiling the programs separately and linking the .o files together once all of the separate compiles have been completed.)

Here is a typical sequence you can use to compile any of the programs in this book:

1. Insert your CLI disk into the internal disk drive. Insert the disk containing your program to compile into the external drive. Then type the command

   ```
   copy hello.c to ram:
   ```

2. When the disk activity light goes out, remove your program disk and insert the Amiga C disk into the external drive. Type the command

   ```
   cd df1:
   ```

   This command tells AmigaDOS that the external drive is to be used as the primary directory for any commands and data files that it needs.

3. Issue the command

   ```
   execute examples/makesimple ram:hello
   ```

With that single command, you have just asked AmigaDOS to use the make program to perform all three phases of the compilation.

AmigaDOS compiles myprogram.c on the ramdisk (the ram: directory) and, if the compilation was successful, produces an executable program file named myprogram (also in the ram: directory). To try the program, you need only type

```
run ram:hello
```

or

```
ram:hello
```

Now, to save the program you have just compiled and linked, you remove the C compiler disk from the external drive, insert your original program disk, and issue the command

```
copy ram:hello to df1:hello
```

Your executable program will be saved to your disk.

Let's look as the contents of a typical make file more closely. By understanding what it does, you will be able to create your own make file for other purposes.

Contents of a Make File

The example make file is actually a command script that AmigaDOS is directed to perform. When you give the command

```
execute <somefilename>
```

AmigaDOS loads its EXECUTE command and takes its input from the file name you specify, just as though you had typed it yourself. Optionally, EXECUTE can perform parameter substitution so that the same command script can be used on different programs, appearing to the computer to be different each time it is run. Files that contain commands for EXECUTE to perform are called execute files.

Execute files can contain several different types of input lines. Among them are comments, parameter substitution commands, and commands that run other programs.

Note that if column 1 of any line contains a period or a semicolon, AmigaDOS treats the line as a comment. The EXECUTE command treats a line beginning with a semicolon as a comment and a line beginning with a period as a command.

Parameter Substitution in an Execute File

You can tell the EXECUTE command that it should prepare to substitute parameters by using a *key statement*. When you provide this statement, EXECUTE makes a copy of your command file in the :T directory on the current disk, takes parameters from your command line immediately following the command name, and substitutes the string value of the parameter in places within your command file where you have requested a parameter substitution.

Consider a command file named typeit, which contains only the following lines:

```
.key firstparm, secondparm
IF EXISTS "<firstparm>"
TYPE "<firstparm>"
ENDIF
IF EXISTS "<secondparm>"
TYPE "<secondparm>"
ENDIF
```

If you type the command

```
execute typeit filenamea filenameb
```

then EXECUTE creates a disk-resident copy of your command file, substituting the first parameter string, here named filenamea, for any occurrence of <firstparm> (the bracketed specification of that parameter name from the key statement) and the second parameter name, filenameb, for <secondparm>:

```
IF EXISTS FILENAMEA
TYPE FILENAMEA
ENDIF
IF EXISTS FILENAMEB
TYPE FILENAMEB
ENDIF
```

This file is then executed as though you typed it at the terminal. This sample type file says: "Check to see if a file exists, and if it does, type it."

The same principle is applied to the C compiler. Use the ED program, or MicroEmacs, or your favorite text editor to look at the contents of examples/make on the Amiga C disk. You will find that the key statement contains parameters for the compiler control commands LC1 and LC2 and a command line for the ALINK command.

Listing B.1 is a typical example of a make file. The line numbers appearing along the left side of this listing are for reference in the explanation that follows. They are *not* part of the file.

Line 1 specifies two parameter names for substitution in the command file: sourcename and listingname. If the file is named make, you run the command file by typing the line

```
execute make myfile
```

to compile the file you have named myfile.c. If the compiler encounters errors, your make file might abort.

Line 6 tests to see if you have provided a source file name. If not, lines 23–25 output instructions. Line 9 tests to see if the file name you have specified exists. If not, line 28 tells you it could not be found.

Lines 13 and 15 provide two alternative methods for specifying the LC1 command, depending on whether there is a listing file name provided. The -i option on the command line shows the compiler where to search for the Include files for the compilation. Include files contain definitions for constants, macros, and data structures that you will need

```
 1  .key sourcename,listingname
 2  ;sourcename is the name of your C language program
 3  ;listingname is the name of the file for the output listing
 4  ; if an output listing is to be produced
 5  ;
 6  IF "<sourcename>" EQ ""
 7  SKIP USAGE
 8  ENDIF
 9  IF NOT EXISTS <sourcename>.c
10  SKIP NOTFOUND
11  ENDIF
12  IF "<listingname>" NOT EQ ""
13      LC1 > <listingname> -iC1.0:include/ <sourcename>.c
14  ELSE
15      LC1 -iC1.0:include/ <sourcename>.c
16  ENDIF
17  ;                   now do compiler phase 2
18  LC2 <sourcename>
19  ;                   then finally link
20  ALINK FROM <sourcename>.o+Lstartup.obj TO <sourcename>
       LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
22  SKIP DONE
21  ;
22  LAB USAGE
23  ECHO "COMMAND IS: make sourcename listingname
24  ECHO "sourcename is mandatory; represents sourcename.c"
25  ECHO "listingname is optional"
26  SKIP DONE
27  LAB NOTFOUND
28  ECHO "<sourcename>.c NOT FOUND!!"
29  LAB DONE
```

Listing B.1: A sample make file

when you use the Amiga system software. The path name to the include files is specified here as "C1.0:include", which means the include directory on your C compiler disk. If you have relabeled the disk using the AmigaDOS RELABEL command, you will have to modify this line to show the actual disk name. Or, for example, you may have C1.1 instead of C1.0. To check which name you should specify, type the CLI command INFO. It will show you the label that is on your disk. It might say, for example:

```
C1.0 [mounted]
```

Line 20 shows one possible form for the ALINK command. This command line links a single file with the startup file and requests that the various libraries be searched for other object code that your program needs to be able to run. If, as is the case for certain of the program segments in this book, you must compile several program segments separately and then link them together, you might want to have a separate version of a make file that does not use line 20. Then use a separate file, perhaps called link, that contains something like the following two lines:

```
.key linkwhat,object
ALINK FROM <linkwhat> + Lstartup.obj TO <object>
   LIBRARY LC.LIB,AMIGA.LIB,DEBUG.LIB
```

Then run the link program by typing the line

```
execute link "fa.o + fb.o + fc.o" outname
```

where the object file names are enclosed in quotes to direct the EXECUTE command to treat this stream of characters as a single parameter for substitution, and outname is the name to be applied to the executable file resulting from the link.

You can find several additional examples of using the EXECUTE command in the *AmigaDOS Users' Manual*.

HOW TO CREATE MAKESIMPLE.A

In Chapter 2, I refer to a file called makesimple.a, which is a derivative of the C-disk examples/make or examples/makesimple program. To create makesimple.a, perform these steps:

1. Copy the makesimple (or make) program to a file named makesimple.a. From the CLI, type:

```
copy df1:examples/makesimple to df1:examples/makesimple.a
```

2. Use a text editor to make the following changes. Change the line

   ```
   :c/lc2 <file>
   ```

   to read

   ```
   :c/lc2 -v <file>
   ```

   This eliminates stack checking code from being added to the compiled program.

3. Change the line that reads

   ```
   :c/alink FROM LIB:LStartup.obj + <file>.o TO <file>
   LIBRARY LIB:lc.lib + LIB:amiga.lib
   ```

   to read

   ```
   :c/alink FROM LIB:AStartup.obj + <file>.o TO <file>  LIBRARY
   LIB:amiga.lib + LIBRARY LIB:lc.lib
   ```

   (Note: Do not press Return until you have typed all of this.) This changes the startup file and the order of use of the library files. It means accepting certain limitations in the library routines provided by amiga.lib. (See the amiga.lib documentation in the *Amiga ROM Kernel Manual* for those limitations.) It also often shrinks the size of the executable file from 12,000 bytes down to as little as 2,000 bytes.

When you use makesimple.a in place of the original make file, your program will link into the amiga.lib version of printf and sprintf, each with the limited formatting capability (most notably no floating-point formatting) instead of the Lattice library version of printf and sprintf. This is one of the reasons that the size of the final executable program is reduced. Note that this could have serious consequences for some types of programs, so be aware of the limitations before you try to use this facility.

As a reminder, the purpose of this make file is to make the code, particularly in Chapter 2, as portable as possible from one compiler version to another (Lattice C, Aztec C, and so on). Thus, rather than use compiler specific functions, such as fopen, fread, and fclose, and compiler specific file handles, such as FILE, Chapter 2 uses Amiga-specific functions, such as Open, Read, and Close, and Amiga-specific file handles that do not correspond to those of any specific compiler.

Readers who are experienced in C programming will likely utilize their own compiler-specific I/O functions. Just be aware that the AmigaDOS file handles are indeed different and cannot be intermixed with those obtained by Unix-like functions provided by the compiler libraries.

Index

Selections from The SYBEX Library

Computer Specific

AMIGA

AMIGA PROGRAMMER'S HANDBOOK Volume 1

by Eugene Mortimore

575 pp., illustr., Ref. 367-8

All the Amiga's power at your fingertips! Organized for working programmers, this is an A to Z compendium of Amiga system facilities, including ROM-BIOS exec calls, the Graphics Library, Animation Library, Layers Library, Intuition calls, and the Workbench.

APPLE II - MACINTOSH

THE PRO-DOS HANDBOOK

by Timothy Rice and Karen Rice

225 pp., illustr. Ref. 230-2

All Pro-DOS users, from beginning to advanced, will find this book packed with vital information. The book covers the basics, and then addresses itself to the Apple II user who needs to interface with Pro-DOS when programming in BASIC. Learn how Pro-DOS uses memory, and how it handles text files, binary files, graphics and sound. Includes a chapter on machine language programming.

PROGRAMMING THE MACINTOSH IN ASSEMBLY LANGUAGE

by Steve Williams

400 pp., illustr. Ref. 263-9

Information, examples, and guidelines for programming the 68000 microprocessor are given, including details of its entire instruction set.

USING THE MACINTOSH TOOLBOX WITH C

by Fred A. Huxham, David Burnard and Jim Takatsuka

559 pp., illustr., Ref. 249-3

In one place, all you need to get applications runnning on the Macintosh, given clearly, completely, and understandably. Featuring the C language.

MASTERING Pro-DOS

by Timothy Rice and Karen Rice

250 pp., illustr., Ref. 315-5

This companion volume to The ProDOS Handbook contains numerous examples of programming techniques and utilities that will be valuable to intermediate and advanced users.

THE EASY GUIDE TO YOUR MACINTOSH

By Joseph Caggiano

214 pp., illustr., Ref. 216-7

Simple and quick to use, this tells first time users how to set up their Macintosh computers and how to use the major features and software.

MACINTOSH FOR COLLEGE STUDENTS

by Bryan Pfaffenberger

250 pp., illustr., Ref. 227-2

Find out how to give yourself an edge in the race to get papers in on time and prepare for exams. This book covers everything you need to know about how to use the Macintosh for college study.

ATARI

UNDERSTANDING ATARI ST BASIC PROGRAMMING

by Tim Knight

300 pp., illustr., Ref. 344-9

Here is a comprehensive tutorial and reference guide for ATARI ST BASIC programming, including graphics, sound and GEM windows. With a complete ST BASIC command summary.

CP/M SYSTEMS

THE CP/M HANDBOOK

by Rodnay Zaks

320 pp., illustr., Ref 048-2

An indispensable reference and guide to CP/M – complete in reference form.
"An excellent reference guide . . ."
Dr. Dobbs Journal

MASTERING CP/M

by Alan Miller

398 pp., illustr., Ref. 068-7

For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system: this book takes up where the CP/M Handbook leaves off.

THE CP/M PLUS HANDBOOK

by Alan Miller

250 pp., illustr., Ref. 158-6

This guide is easy for beginners to understand, yet contains valuable information for advanced users of CP/M Plus.

MASTERING DISK OPERATIONS ON THE COMMODORE 128

by Alan R. Miller

f238 pp., illustr., Ref. 357-0

This guide to using CP/M Plus on the Commodore 128 is essential for users at all levels, offering introductory tutorials, in-depth treatment of major topics, a look inside the operating system, and a CP/M Plus command summary.

IBM PC AND COMPATIBLES

OPERATING THE IBM PC NETWORKS

Token Ring and Broadband

by Paul Berry

363 pp., illustr., Ref. 307-4

This tells you how to plan, install, and use either the Token Ring Network or the PC Network. Focusing on the hardware-independent PCN software, this book gives readers who need to plan, set-up, operate, and administrate such networks the head start they need to see their way clearly right from the beginning.

THE ABC'S OF THE IBM PC

by Joan Lasselle and Carol Ramsay (2nd Edition)

200 pp., illustr., Ref. 370-8

Complete hands-on training for first-time users—in clear, understandable terms. With step-by-step tutorials on everything from handling disks, to running programs, to using the PC's special capabilities.

MS-DOS POWER USER'S GUIDE

by Jonathan Kamin

400 pp., illustr., Ref. 345-7

A guide to the advanced and subtle features of DOS. Contains a goldmine of techniques to streamline operations by automating complex tasks and repeated operations. Includes a review of the basics, plus tutorials on less familiar DOS functions. For version 2.1 through 3.1

THE MS-DOS HANDBOOK

by Richard Allen King (2nd Ed)

320 pp., illustr., Ref. 185-3

The differences between the various versions and manufacturer's implementations of MS-DOS are covered in a clear straightforward manner. Tables, maps, and numerous examples make this the most complete book on MS-DOS available.

ESSENTIAL PC-DOS

by Myril and Susan Shaw

300 pp., illustr., Ref. 176-4

Whether you work with the IBM PC, XT, PC jr. or the portable PC, this book will be invaluable both for learning PC DOS and for later reference.

THE IBM PC-DOS HANDBOOK

by Richard Allen King

296 pp., Ref. 103-9

Explains the PC disk operating system. Get the most out of your PC by adapting its capabilities to your specific needs with confidence. Includes both the PC-DOS features and functions, and also the advanced capabilities.

BUSINESS GRAPHICS FOR THE IBM PC

by Nelson Ford

259 pp., illustr. Ref. 124-1

Ready-to-run programs for creating line graphs, multiple bar graphs, pie charts and more. An ideal way to use your PC's business capabilities!

MASTERING THINKTANK ON THE IBM PC

by Jonathan Kamin

350 pp., illustr, Ref. 327-9

This comprehensive guide to idea processing with ThinkTank takes you from starting a first outline to mastering advanced features. It includes undocumented tips and tricks and an introduction to *Ready!*, the RAM-resident outline processor.

THE IBM PC CONNECTION

by James Coffron

264 pp., illustr., Ref. 127-6

Teaches elementary interfacing and BASIC programming of the IBM PC for connection to external devices and household appliances.

DATA FILE PROGRAMMING ON YOUR IBM PC

by Alan Simpson

219 pp., illustr., Ref. 146-2

This book provides instructions and examples for managing data files in BASIC Programming. Design and development are extensively discussed.

Software Specific

SPREADSHEETS

UNDERSTANDING JAVELIN

by John R. Levine, Margaret H. Young, and Jordan M. Young

350 pp., illustr., Ref. 358-9

A complete guide to Javelin, including an introduction to the theory of modeling. Business-minded examples show Javelin at work on budgets, graphs, forecasts, flow charts, and much more.

MASTERING SUPERCALC 3

by Greg Harvey

300 pp., illustr., Ref. 312-0

Featuring Version 2.1, this title offers full coverage of all the sophisticated features of this third generation spreadsheet, including spreadsheet, graphics, database and advanced techniques.

DOING BUSINESS WITH MULTIPLAN

by Richard Allen King and Stanley R. Trost

250 pp., illustr., Ref. 148-9

This book will show you how using Multiplan can be nearly as easy as learning to use a pocket calculator. It presents a collection of templates for business applications.

MULTIPLAN ON THE COMMODORE 64

by Richard Allen King

250 pp., illustr. Ref. 231-0

This clear, straightforward guide will give you a firm grasp on Multiplan's function, as well as provide a collection of useful template programs.

WORD PROCESSING

INTRODUCTION TO WORDSTAR (3rd Edition)

by Arthur Naiman

208 pp., illustr., Ref. 134-9

A bestselling SYBEX classic. "WordStar is complicated enough to need a book to get you into it comfortably. Naiman's **Introduction to WordStar** is the best."

—Whole Earth Software Catalog

" . . . an indespensable fingertip guide, highly recommended for beginners and experienced users."

—TypeWorld

PRACTICAL WORDSTAR USES

by Julie Anne Arca

303 pp., illustr. Ref. 107-1

Pick your most time-consuming wordprocessing tasks and this book will show you how to streamline them with WordStar.

MASTERING WORDSTAR ON THE IBM PC

by Arthur Naiman

200 pp., illustr., Ref. 250-7

The classic Introduction to WordStar is now specially presented for the IBM PC, complete with margin-flagged keys and other valuable quick-reference tools.

WORDSTAR TIPS AND TRAPS

by Dick Andersen, Cynthia Cooper, and Janet McBeen

300 pp., illustr., Ref. 261-2

The handbook every WordStar user has been waiting for: a goldmine of expert techniques for speed, efficiency, and easy troubleshooting. Arranged by topic for fast reference.

THE COMPLETE GUIDE TO MULTIMATE

by Carol Holcomb Dreger

250 pp., illustr. Ref. 229-9

A concise introduction to the many applications of this powerful word processing program, arranged in tutorial form.

PRACTICAL MULTIMATE USES

by Chris Gilbert

275 pp., illustr., Ref. 276-0

Includes an overview followed by practical business techniques, this covers documentation, formatting, tables, and Key Procedures.

MASTERING DISPLAYWRITE 3

by Michael McCarthy

447 pp., illustr., Ref. 340-6

A complete introduction to full-featured word processing, from first start-up to advanced applications—designed with the corporate user in mind. Includes complete appendices for quick reference and troubleshooting.

MASTERING WORDPERFECT

by Susan Baake Kelly

397 pp., illustr., Ref. 332-5

Solid training and support for every WordPerfect user—with concise tutorials, thorough treatment of advanced features and "recipes" for business uses. Covers all versions through 4.1.

WORDPERFECT TIPS AND TRICKS

by Alan R. Neibauer

350pp., illustr., Ref. 360-0

A practical companion for users of WordPerfect versions through 4.1—packed with clear explanations and "recipes" for creative uses, including outline processing, graphics, spreadsheet and data management.

MASTERING SAMNA

by Ann McFarland Draper

425 pp., illustr., Ref. 376-7

Learn the power of SAMNA Word and the SAMNA spreadsheet from an expert user and teacher. This comprehensive tutorial lets you build on the basics to get the most from the software's unique features.

MASTERING MS WORD

by Mathew Holtz

365 pp., illustr., Ref. 285-X

This clearly-written guide to MS WORD begins by teaching fundamentals quickly and then putting them to use right away.

Covers material useful to new and experienced word processors.

PRACTICAL TECHNIQUES IN MS WORD

by Alan R. Neibauer

300 pp., illustr., Ref. 316-3

This book expands into the full power of MS WORD, stressing techniques and procedures to streamline document preparation, including specialized uses such as financial documents and even graphics.

INTRODUCTION TO WORDSTAR 2000

by David Kolodnay and Thomas Blackadar

292 pp., illustr., Ref. 270-1

This book covers all the essential features of WordStar 2000 for both beginners and former WordStar users.

PRACTICAL TECHNIQUES IN WORDSTAR 2000

by John Donovan

250 pp., illustr., Ref. 272-8

Featuring WordStar 2000 Release 2, this book presents task-oriented tutorials that get to the heart of practical business solutions.

MASTERING THINKTANK ON THE 512K MACINTOSH

by Jonathan Kamin

264 pp., illustr., Ref. 305-8

Idea-processing at your fingertips: from basic to advanced applications, including answers to the technical question most frequently asked by users.

DATABASE MANAGEMENT SYSTEMS

UNDERSTANDING dBASE III PLUS

by Alan Simpson

415 pp., illustr., Ref. 349-X

Emphasizing the new PLUS features, this extensive volume gives the database terminology, program management, techniques, and applications. There are hints on file-handling, debugging, avoiding syntax errors.

ADVANCED TECHNIQUES IN dBASE III PLUS

by Alan Simpson

500 pp., illustr., Ref. 369-4

The latest version of what ***Databased Advisor*** called "the best choice for experienced dBASE III programmers." Stressing design and structured programming for quality custom systems, it includes practical examples and full details on PLUS features.

MASTERING dBASE III PLUS: A STRUCTURED APPROACH

by Carl Townsend

350 pp., illustr., Ref. 372-4

This new edition adds the power of PLUS to Townsend's highly successful structured approach to dBASE III programming. Useful examples from business illustrate system design techniques for superior custom applications.

ABC'S OF dBASE III PLUS

by Robert Cowart

225 pp., illustr., Ref. 379-1

Complete introduction to dBASE III PLUS for first-time users who want to get up and running with dBASE fast. With step-by-step exercises covering the essential functions as well as many useful tips and business applications.

UNDERSTANDING dBASE III

by Alan Simpson

250 pp., illustr., Ref. 267-1

The basics and more, for beginners and intermediate users of dBASEIII. This presents mailing label systems, bookkeeping and data management at your fingertips.

ADVANCED TECHNIQUES IN dBASE III

by Alan Simpson

505 pp., illustr., Ref. 282-5

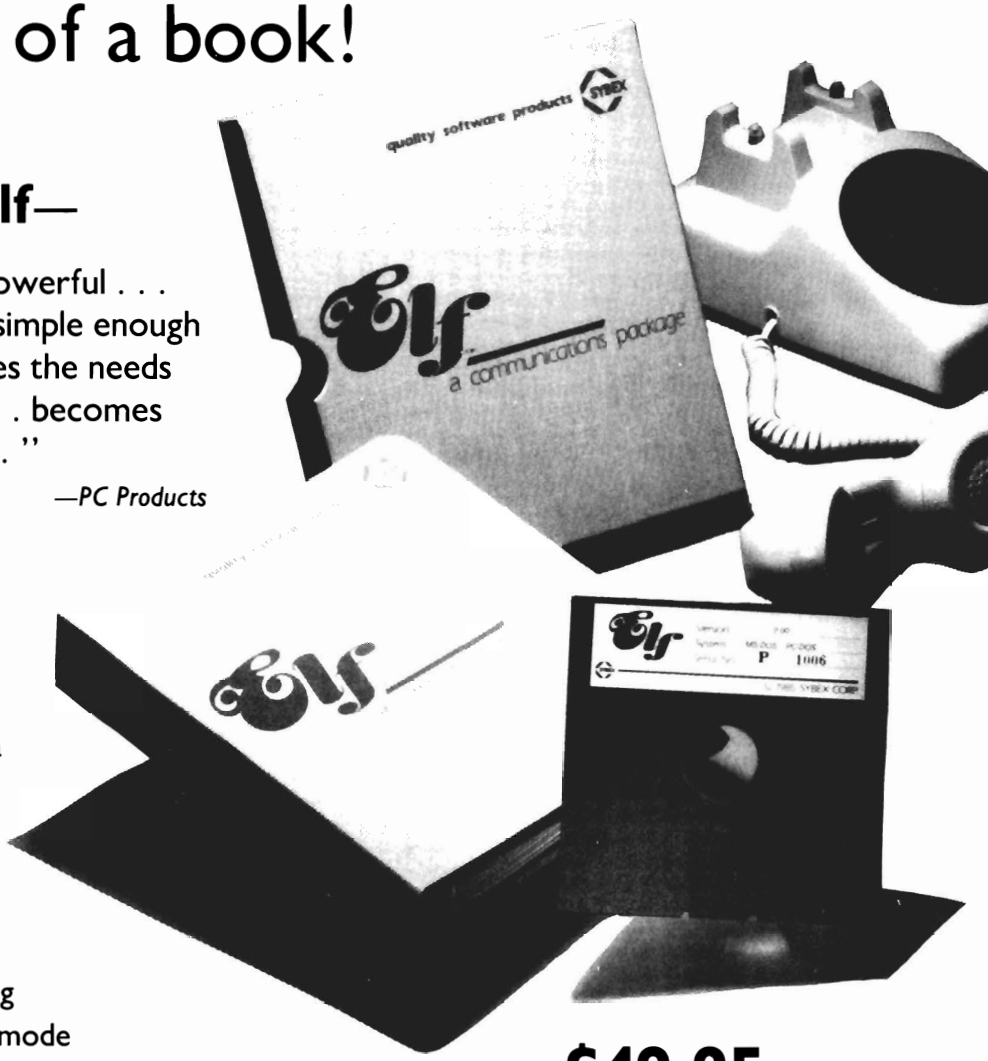
Intermediate to experienced users are given the best database design techniques, the primary focus being the development of user-friendly, customized programs.

SYBEX introduces one of the most sophisticated communications packages for your IBM-PC at the cost of a book!

Connect with **Elf**—

"as easy to use as it is powerful . . . easy to understand . . . simple enough for beginners . . . satisfies the needs of sophisticated users . . . becomes practically automatic . . . "

—PC Products



Elf automates virtually all calls and file transfers and turns your computer into a remote.

- Menu driven
- Easy to install
- Personality modules
- Crash-proof error handling
- Password protected post mode
- Supports XMODEM protocol
- Includes SYBEX-quality documentation

$49.95
(Diskette and Book)

ORDER NOW! CALL: 800-227-2346

Programmer's Guide to the Amiga

Programs Available on Disk

If you'd like to use the programs in this book but don't want to type them in yourself, you can send for a disk containing all the programs in this book. To obtain this disk, complete the order form and return it along with a check or money order for $15.00. California residents add sales tax.

DATAPATH
P.O. Box 1828
Los Gatos, CA 95031-1828
(408) 353-3901

Name_____

Address_____

City/State/ZIP_____

Enclosed is my check or money order.
(Make check payable to *DATAPATH*)

Programmer's Guide to the Amiga

SYBEX is not affiliated with DATAPATH *and assumes no responsibility for any defect in the disk or program.*

SYBEX Computer Books are different.

Here is why . . .

At SYBEX, each book is designed with you in mind. Every manuscript is carefully selected and supervised by our editors, who are themselves computer experts. We publish the best authors, whose technical expertise is matched by an ability to write clearly and to communicate effectively. Programs are thoroughly tested for accuracy by our technical staff. Our computerized production department goes to great lengths to make sure that each book is well-designed.

In the pursuit of timeliness, SYBEX has achieved many publishing firsts. SYBEX was among the first to integrate personal computers used by authors and staff into the publishing process. SYBEX was the first to publish books on the CP/M operating system, microprocessor interfacing techniques, word processing, and many more topics.

Expertise in computers and dedication to the highest quality product have made SYBEX a world leader in computer book publishing. Translated into fourteen languages, SYBEX books have helped millions of people around the world to get the most from their computers. We hope we have helped you, too.

For a complete catalog of our publications:

SYBEX, Inc. 2021 Challenger Drive, #100, Alameda, CA 94501
Tel: (415) 523-8233/(800) 227-2346 Telex: 336311

PROGRAMMER'S GUIDE TO THE AMIGA

The **Programmer's Guide to the Amiga** is a guided hands-on tour through the Amiga system, packed with in-depth information and sample programs you won't find anywhere else. This book is a must for programmers, owners, and software developers—anyone who wants to master and use the Amiga's unique capabilities.

Chapter One gives an overview of the Amiga system organization, with subsequent chapters devoted to all its major components—AmigaDOS, Exec, Graphics, Intuition, Devices, Sound, Graphics Animation, and more. You'll find:

- unique, in-depth treatment of terminal-based programming on the Amiga, with full details on the DOS functions and filing system
- step-by-step programming examples in every chapter, illustrating proper use of system routines
- ready-to-use routines that simplify programming for superb graphics, animation, device control, managing the user interface, and much more
- an introduction to multitasking and message-passing procedures
- complete appendices on compiling programs and using the program text editor

This guide will be invaluable to every Amiga programmer—especially those working in C, Pascal, and assembly language. All sample programs are in Amiga C.

"Definitely worth its cover price and then some."
—Commodore

"This book is clear and accurate, with plenty of C language programming examples."
—COMPUTE

"Plenty of valuable information."
—Ahoy!'s AmigaUser

About the Author

Robert A. Peck, an engineer and programmer, headed the Amiga documentation team. He has written numerous computer manuals, tutorials, and reference guides, including the *Amiga Hardware Manual* and the *Amiga ROM Kernel Manual*, as well as articles for such publications as *Byte, Kilobaud, Micro, Compute!,* and *Robotics Age.* He lives in Los Gatos, California.

SYBEX books bring you skills—not just information.
As computer experts, educators, and publishing professionals, we care—and it shows.
You can trust the SYBEX label of excellence.

SYBEX

ISBN 0-89588-310-4 U.S. $24.95