

COMPUTE!'s

ST Applications GUIDE

PROGRAMMING IN C

Simon Field,
Kathleen Mandis,
and Dave Myers

A clear and comprehensive
guide to writing Atari ST
applications in C.



Contents

Foreword

COMPUTE!'s ST Applications PROGRAMMING IN C

1. Creating Menus, Dialog Boxes, and Graphics	85
2. Building a Command Post-It Desk Accessory	129
3. Changing a Desk Accessory to a Regular Program	155
4. Programming the Sound Chip	175
5. A Debugging Aid	211
6. A Glossary	231

Simon Field,
Kathleen Mandis,
and Dave Myers

COMPUTE! Publications, Inc. 

Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies

Greensboro, North Carolina

Copyright 1986, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-078-5

The authors and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information. However, the information and programs in this book are sold without warranty, either express or implied. Neither the authors nor COMPUTE! Publications, Inc. will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the programs or information in this book.

The opinions expressed in this book are solely those of the author and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Atari, Atari 520ST, Atari 1040ST, and TOS are trademarks or registered trademarks of Atari Corporation. GEM is a trademark of Digital Research, Inc.

Contents

Foreword	v
Introduction	vii
1. C Programming and the Atari ST	1
2. Creating the GEM Programming Envelope	9
3. Simple Line Graphics	47
4. Business Graphics	59
5. Creating Menus, Dialog Boxes, and Graphics	85
6. Building a Command Shell Desk Accessory	125
7. Changing a Desk Accessory to a Regular Program	165
8. Programming the Sound Chip	173
9. A Debugging Aid	221
10. A Disassembler	261
Appendix: World Map Data	301
Index	322
Disk Coupon	327



Foreword

■ The Atari ST has a user interface called GEM. GEM's features include pull-down menus, icons, sliders to scroll data, mouse-activated screen selection, the ability to move and rearrange windows, and so on. In short, GEM is how users get the ST to work. But how do programmers get GEM features to work in their application programs?

■ *COMPUTE!'s ST Applications Guide: Programming in C* is written for C programmers. It does *not* teach C programming, but rather includes a library of functions that makes using GEM routines easier to access from your own C programs. Each function in the library is fully explained and easy to use.

The book begins by introducing the library of routines, called the *envelope library*. You can write application programs within the envelope and then use the envelope's functions to access GEM and make its features part of your program. The rest of the book gives you a series of example application programs that use the envelope library to interface with GEM.

Examples of the application programs included are a Mandelbrot set of fractal graphics that takes only minutes, rather than hours, to draw completely; a Command Shell which allows you to execute the commands copy, move, remove, print, list (print with filename), dir, and chdir from a desk accessory; a debug program that will help you analyze the reason a program crashed; plus a sound program and graphing applications.

COMPUTE!'s ST Applications Guide: Programming in C requires a working knowledge of C programming and a C compiler for your Atari ST. All the programs are ready to type in and use. Also available, on a double-sided 3½-inch disk, are all the programs from the book. To order the disk, use the coupon in the back of the book or call 1-800-346-6767 (in New York 212-887-8525).



Introduction

Many programmers learn best by example. As with music, where no amount of discussion is as good as hearing a performance, seeing examples of good programming is much more useful than reading dry descriptions of functions and interfaces.

Starting with that premise, this book does not simply describe a function and list its arguments, leaving you to puzzle over how it is meant to be used. That's the realm of reference books and dictionaries. Instead, it discusses the insides of programs, showing how functions fit together to create working applications for business, art, graphics, and music.

To use this book you'll need an understanding of C programming. *This book is written for programmers who know the C programming language*—it is not intended for the beginning C programmer.

The programs in this book are about the programming interface—known as GEM—on the Atari 520 and 1040ST. GEM is a rich environment, with hundreds of routines enabling you to program sophisticated, powerful applications. GEM's features include pull-down menus, icons, sliders to scroll screen data, mouse-activated screen selections, the ability to move and rearrange windows, and so on. But that richness has a price. GEM can be complicated and intimidating, hard for all but the most dedicated to program. How do you, the programmer, get GEM features to work in your applications programs? That's what this book is about.

Much of the information available about GEM comes from the *Atari ST Software Developer's Kit*, which consists of 400-plus pages of technical notes, and which costs about \$300. Although the Kit is designed for developers, as it is now organized, vital information is dispersed in various nooks, crannies, and dark corners of the notes. Even more serious is the lack of a clear "system" for GEM programming in the sense that many procedures needed to program GEM are either not readily identifiable or are described at a low level of programming. Finally, those procedures and routines that are well documented may contain errors or other quirks you should know about.

This book rectifies those problems. We build a set of high-level routines that simplify working with GEM by avoiding the errors and automating much of GEM's complexity. The routines make the computer take care of the ordinary trivia, and let you concentrate on your applications programming. Think of the set of routines as an *envelope* around GEM. You can write applications programs within the envelope and then use the envelope's routines to access GEM, coupling its features to your program.

At its simplest level, the envelope provides so much of the user interface that your application program may never need to interact directly with the user at all. Whether responding to items selected from a menu, typed characters, or mouse moves, the envelope calls the appropriate routines to handle the input. Moreover, all procedures to manage windows or refresh the screen are done automatically.


Each of the programs in this book uses the envelope routines. Rather than just presenting the programs, however, we take them apart to show how their envelope routines control GEM. Piece by piece, the programs unfold to show how the routines interconnect to create a realistic, working application.

Use the routines, modified or not, in your own programs. They were written with the idea that you'll want to lift sections of code directly and put them into other routines or programs. You can also modify the routines to create new ways of working with GEM, knowing beforehand what works.

The information in this book comes from many hours of testing and evaluating the routines in the *Atari ST Software Developer's Kit*. You do *not* need the Kit for programming with GEM. All of the currently available C compilers for the ST include the libraries and routines necessary for the envelope and for writing ST applications programs. But, if you don't have the Developer's Kit, you *will* need one of the other C compilers. *Some of the routines' names may differ with the various compilers, but the changes should be noted in the compilers' documentation.* If you use another compiler, especially one that cannot link programs with more than 32K of data, you'll need to make some additional changes to the routines included in this book; please see the last section of Chapter 1 for instructions.

By the way, for simplicity we use the term GEM to refer to all of the software in the ST's ROM and subroutine libraries. If a subroutine or macro is part of the BIOS, the VDI, the AES, or GEMDOS, we lump it in the class GEM because the differences are usually irrelevant for what we're doing in this book. The important distinction is between the routines available to everyone from Atari or the C compiler, and the routines available only in this book.

This book builds on itself after Chapter 1. That is, the procedures developed in Chapter 2 are essential for Chapter 3; those in Chapter 3 for Chapter 4, and so on. The biggest, and most important, chapter is Chapter 2. It is there that we describe the envelope that is used in every subsequent applications program in the book. Therefore, once you get through with Chapter 2, you might be tempted to write your own ST applications program right away. Resist for a while anyway. Later chapters reveal many simplified programming techniques and may solve some vexing problem you're facing. Then, by the end of the book, you'll be able to write an ST applications program with a minimum of fuss.



1 C Programming and the Atari ST



1 C Programming and the Atari ST

How a programmer writes a program can often have quite an effect on the success of the program. If a program is easy to read, unambiguous, and modular, then it is easier to debug, easier to maintain, easier to build and understand, and easier to share with other programmers. While style is often a personal matter, and discussions of where to place C's curly braces often sound like religious arguments, in practice, if a programmer's style is consistent, then a reader can learn to read it with little trouble.

In this book, the programs were written in a style designed to make programming errors less likely to occur, and to make the code as readable and understandable as possible. Some aspects of this style may be summarized as follows.

1. Indentation is rigorously used to clearly mark which control statements govern each block of code. Control statements like *if* and *while* always control a statement or group of statements that is indented on the following line, not on the same line. Thus you can use

```
if( index > MAX_INDEX )  
    return;
```

instead of placing the *return* on the same line as the *if*. Doing otherwise can lead the reader to think that the wrong statement is being controlled, as in

```
while( *p++ = *q++ )  
    ;  
    putchar( '\n' );
```

where, if the semicolon was not on a line by itself, the reader might think the *putchar* was controlled by the *while*.

2. Modules are used to make code more understandable and to make the code re-usable. Functions are, in general, kept short and given meaningful names, so that the reader can think about them clearly in terms of their function, instead of wrestling with routines that do several different jobs while wearing one hat.

Some of the longer routines in this book are based on the C switch statement. Each case in the switch acts like a named subroutine, in that the

code is broken down into manageable parts and *named* by the case it handles.

Re-usability means that, once a function has been written, it can be used for more than one purpose or used by more than one program. This not only makes new programs quicker to write, but it makes them less error-prone, and more understandable. If a function has been fully debugged in one program, it's less likely to introduce problems in another program than a function which has been written from scratch. If the person reading the code has seen a function before in another program, it's easier for him to understand the new program, since he is on familiar ground, with less new information to learn.

3. Global variables are used sparingly. In this book, globals are used carefully, and are hidden from functions that do not need to know about them. There are no header files full of external statements, but instead each function declares its own external variables to keep the information close by, where the reader can refer to it easily, and to make it obvious which functions use a particular global variable.

The concept of locality helps keep the reader's mind on the task at hand. If the interface between a subroutine and the outside world is clearly defined, then the subroutine can be understood by itself. Global variables muddle this interface, and the reader must know the current state of some other part of the program in order to understand the subroutine at hand.

4. Braces are placed consistently throughout the book. Open curly braces are placed at the end of the control statement that governs a block, and end curly braces are on a line by themselves, indented with the block of statements they terminate. Other styles that are popular have the open brace on a line by itself, or have the end brace indented at the level of the control statement. If the programmer is consistent, then any of these styles is readable.
5. The use of the goto statement is another "religious" issue. None are used in this book, but only because none were needed. The use of a goto should be a flag to the programmer that a function should be split into two functions to make the code more readable. In cases where speed is of paramount importance (which is actually quite rare) a well-commented goto statement might be the best choice.

The goto statement should not be used where structured statements such as *while* or *break* could be used instead. However, since C cannot use the break statement to get out of a nested loop, a goto is often used when this needs to be done. When a few extra microseconds will not be missed, however, it is usually better to put the inner loop in a separate subroutine, and end the outer loop if the subroutine returns a FALSE when called.

Using Libraries

Libraries of functions are a logical outgrowth of the need for modularity and re-usability. Libraries allow the programmer to save useful functions in a place where the linker can find them, and to include them in any program that needs their functionality. If a program needs a slightly different version of a function in the library, the programmer can merely include the altered function in the list of files given to the linker, and the linker will use the new version, and not get the old version from the library.

Such flexibility is used throughout this book to make programming the ST remarkably easy. As you develop your own functions, you are encouraged to make them generic and re-usable, and to put them in a library where they will be easily accessible for future programs.

Programming on the ST

The programs in this book were written with *Alcyon C*, the C compiler included in the *Atari ST Software Developer's Kit*, although other compilers should present no problem since the programs adhere strictly to the Kernighan and Ritchie standard for C. In the two last chapters, a very small amount of machine language is used when the code gets very machine-specific in dealing with issues of debugging, but converting to another assembler should be trivial.

The *Atari ST Software Developer's Kit* compiler is somewhat cumbersome to use, since there is no single command that will compile a C program. The compiler comes in pieces, and the programmer must use a batch program to execute the parts of the compiler in sequence from a script file. Atari supplies such a batch file, called *C.BAT*, to be used to compile programs. The *C.BAT* file produces object files, which are then linked by another batch file which calls the linker and a post processor called *relmod.prg*.

Each program in this book will be presented with a batch file that shows how to link the program, and which libraries are needed to link with the file. The batch files are specific to the *Atari ST Software Developer's Kit*, but can be used as guidelines for other compilers and linkers, since library names are surprisingly consistent among compilers. To make conversion easier, the library names and a description of their contents are given here:

Name	Contents
gemlib	printf, open, sprintf, etc. (C library)
aesbind	wind_get, evnt_multi, etc. (AES library)
vdibind	v_pline, v_circle, etc. (VDI library)
libf	fpadd, fpmul, etc. (floating-point library)
osbind.o	gemdos(), bios(), and xbios() routines

Modifications to Atari-Supplied Routines

If you are using the *Atari ST Software Developer's Kit* to compile the programs in this book, you will need to make some modifications to two of the routines Atari supplies. These are startup routines that need to be linked as the first thing in any program. The `accstart.o` routine is the one you choose for accessories and the `gemstart.o` routine is for regular programs.

Our reason for recommending that you modify these routines is that they define very small amounts of stack space (by normal C programming standards). Good C programs minimize the use of global variables, and instead make use of local variables on the program's stack. The two startup routines can be modified so they define about 32K of stack, which is more than adequate for most programs. If you write programs that need more space, you can increase the size with the following procedure.

In the `GEMSTART.S` file on the *Developer's Kit* disk is some code that looks like this:

```

move.l  a7,a5      * save a7 so we can get the basepage address
move.l  4(a5),a5    * a5=basepage address
move.l  a5,_base    * save for C startup
move.l  $(a5),d0
add.l   $14(a5),d0
add.l   $1c(a5),d0
add.l   #$500,d0    * d0=basepage+textlen+datalen+bsslen (plus 1K of user
                    stack)

move.l  d0,d1
add.l   a5,d1      * compute stack top
and.l   #-2,d1     * insure even byte boundary
move.l  d1,a7      * set up user stack, 1K above end of BSS

```

You need to change the `#$500` to `#$8100` as is done in the following code. The comments have also been changed to reflect the code change.

```

move.l  a7,a5      * save a7 so we can get the basepage address
move.l  4(a5),a5    * a5=basepage address
move.l  a5,_base    * save for C startup
move.l  $(a5),d0
add.l   $14(a5),d0
add.l   $1c(a5),d0
add.l   #$8100,d0   * d0=basepage+textlen+datalen+bsslen (plus 32K of user
                    stack)

move.l  d0,d1
add.l   a5,d1      * compute stack top
and.l   #-2,d1     * insure even byte boundary
move.l  d1,a7      * set up user stack, 1K above end of BSS

```

The Atari version of ACCSTART.S defines only 256 bytes of stack. You can redefine the stack space to 32K bytes with the following changes. First, find the lines in the ACCSTART.S file that look like this:

```
.bss
.even
retsav: .ds.l 1
        .ds.l 256
ustk:   .ds.l 1
```

and change the 256 to 32768 like this:

```
.bss
.even
retsav: .ds.l 1
        .ds.l 32768
ustk:   .ds.l 1
```

Last, you need to assemble the files ACCSTART.S and GEMSTART.S using the as68.prg program supplied with the *Developer's Kit*. From command.tos, give the following commands:

```
as68 -l -u accstart.s
as68 -l -u gemstart.s
```

The option `-l` tells the assembler to make all address constants 32 bits instead of 16 bits, and the `-u` option causes any undefined variables to be declared as global variables. The files ACCSTART.O and GEMSTART.O are produced.

Other Compilers

If your compiler cannot link programs with more than 32K of data, then some of the large arrays used in the programs in this book will have to be changed to use the `malloc ()` subroutine to allocate the memory at runtime. Some of the programs declare arrays to hold copies of the ST's screen (which is 32,000 bytes long), and some others declare arrays that hold text to be printed (200 lines of 80 characters each). Using `malloc ()` is slightly less efficient, and more cumbersome, but necessary for some compilers not designed to take advantage of the ST's 512K (or more) of memory.

The procedure for using the `malloc ()` subroutine to allocate memory for an array in excess of 32,000 bytes at runtime is to change the array into a pointer to an array. For example, the line

```
long int screen[200][40];
```

will become


```
long int *screen = 0;
```

CHAPTER 1

Then, before the array (now a pointer) is first referenced, some code like

```
if ( screen == 0 )  
    screen = (long int *) malloc ( 200 * 40 * 4 );
```

must be added to initialize the pointer to point to 32,000 bytes of memory freshly allocated by the malloc routine.



2 Creating the GEM Programming Envelope



2 Creating the GEM Programming Envelope

■ GEM provides a large number of basic operations for controlling the computer and creating simple-to-use, graphic user interfaces. Although GEM is somewhat complicated to use, the sophisticated results you can achieve with it are evidence of its quality as a programming tool. To make programming easier, you can construct higher-level routines that deal with GEM.

This chapter simplifies the task of creating new programs with GEM by creating a set of routines for dealing with GEM on a higher level. You'll be able to use these routines with any C program; the chapter will explain how to set the appropriate variables in the routines to change certain GEM interface features to suit your particular application program. In effect, your C program code will be surrounded by these routines, which take care of windows, mice, keyboard input, and messages from GEM.

This set of routines which surrounds your program is referred to as the *envelope library*, or just *envelope*.

Since the envelope library is general-purpose and knows little about the program it surrounds, you must provide some routines that connect your program with the higher-level envelope routines. These connecting routines are:

<code>got_key</code>	Called when a key is pressed
<code>mouse_hit</code>	Called when a mouse button is pressed
<code>doit</code>	Called when the screen needs refreshing
<code>open_data</code>	Deals with input files
<code>build_tree</code>	An alternative to RCS for menus
<code>do_menu</code>	Called when a menu item has been selected

You can change the default versions provided for these routines according to the functions your program needs, such as menus, or mouse and keyboard input.

One more file is necessary for the envelope. That file—`config.c`—tailors,

or configures, the envelope's behavior and appearance to a particular application. The config.c file is explained below.

Conceptually, when you execute your application program in the GEM environment, the various routine sets will look like Figure 2-1.

Figure 2-1. Your C Program and the Envelope Library

Application Program
Connecting Routines, config. C
Envelope Library
GEM
TOS

The config.c File

The config.c file contains the settings for interface variables such as the window name and size and how the program should react to keyboard and mouse input. Not all the variables in Program 2-1 are used by every program; for example, some only apply to a desk accessory and are ignored. However, the config.c file is designed to contain all the variables that need to be individually set for each application program.

The first line in config.c specifies the name that is to appear in the top of a program window. The variable is called `wind_name` and in the example file the name of the window is Command Shell.

Next, to make it convenient to choose between building menus with the GEM Resource Construction Set or to build menus dynamically using our own functions, the constant `USE_RCS` is used to initialize the variable named `resource`. If `resource` is 0, we'll build our own menus; if it is set to a filename, we'll read menus from a file. In our example, if you type

```
#define USE_RCS = 1
```

as the first line in the config.c file, the menus will be read from a file named SHELL.RSC.

Next, we'll define variables used to define a particular program as a desk accessory application. Setting up the envelope library in this fashion means it will work for either type of program—a desk accessory or a regular program—without modification. The config.c file is the only part that changes. The variable `access_name` is set to Command Shell, the name which will appear in the Desk menu on the desktop if this program is a desk accessory. To define this program as a desk accessory, the `i_am_accessory` is defined as 1; otherwise it is 0 for a program.

Since desk accessories usually appear in windows that are smaller than full-screen, window size and placement need to be defined. These small window variables are only used if `i_am_accessory` is set to 1. If `i_am_accessory` is set to something other than 1, the variables that set the window to a smaller size are ignored. The variables used are `sx`, `sy`—for the *x* and *y* coordinates—and `sh`, `sw`—for the height and width. The initial positions of the slider boxes within the window slider areas are controlled by `slv`, `slh`, `svs`, and `shs`.

Usually, there is a minimum useful size for an application window. The `min_high` and `min_wide` variables control how small a user can size the program window.

For programs that use time intervals, such as some games or a clock, the interval variable defines the milliseconds between messages from GEM that a Timer Event has occurred. If `interval` is set to 0, no timing occurs.

GEM manages the screen windows and input devices such as the mouse and keyboard. When GEM detects an event, such as a mouse click or a window being closed or resized on the screen, it will send a message about the event. The event variable is set to the constants as defined in the *GEM Software Developer's Kit*, produced by Digital Research Incorporated, that represent the events you want for your program. `MU_MESAG` lets the envelope receive GEM messages—for example, a message that it's time to repaint the screen. `MU_BUTTON` and `MU_KEYBD` let the envelope receive mouse clicks and keypresses, respectively. `MU_M1` and `MU_M2` allow the envelope to receive a message if the mouse enters a rectangular area you've defined.

This configuration file serves all of our purposes for the library of envelope routines in this book. Since the envelope produces a fairly robust use of the GEM interface capabilities, and you can replace our C programs with those of your own, our `config.c` example file may be adequate for your needs. However, our example should give you the idea of how to create your own configuration file if you should add new variables or modify the envelope routines.

Program 2-1. `config.c`

```
# include <gemdefs.h>

char *wind_name          = " Command Shell ";

#ifdef USE_RCS
char *resource            = "SHELL.RSC";
#else
char *resource            = 0;
#endif USE_RCS

char *access_name        = " Command Shell ";
int i_am_accessory       = 1;
int sx                   = 20; /* small window size */
int sy                   = 50;
int sw                   = 250;
int sh                   = 125;
int slv                  = 0; /* small window vertical slider pos */
int slh                  = 0; /* small window horizontal slider pos */
int svs                  = 1000; /* small window vertical slider size */
```

CHAPTER 2

```
int shs           = 1000; /* small window horizontal slider size */
int min_wide      = 100;
int min_high      = 50;
int interval      = 30000;
int events = MU_MESAG : MU_BUTTON : MU_KEYBD : MU_M1 : MU_M2;
```

The main.c Routine

The first routine in the library is main, Program 2-2. main will set up the screen so that what is written appears in the GEM desktop interface. The function calls in main will set up a window with sliders, a title, and so on; handle all the keyboard and mouse input; change the mouse cursor to a pointing finger; hide the TOS cursor; and then exit.

The main routine sets up the screen by calling the `setup_screen` function (described in detail later), which returns an integer used by GEM to identify the *virtual workstation*. There can be several virtual workstations, and each time `setup_screen` is called it returns a different one, but we'll use only one virtual workstation in our program.

The *virtual workstation* is a concept that allows both you and GEM to perform graphics operations without knowing what the display device is. By using a virtual workstation, you can use the same commands to draw on a plotter as for the screen, even though the pixel resolution, color capabilities, and drawing methods may be totally different. At display time, GEM does the appropriate display technique for the device. For example, for printers or plotters, GEM may buffer commands so plotting can be done without having to reverse the paper. On the screen, the commands are executed immediately and quickly. When the program says to clear the virtual workstation, GEM issues a command to the actual device, to either clear the screen or eject a page.

The integer that is returned by the `setup_screen` call is stored as the virtual workstation *handle* in the variable `vw_hand`. This handle is passed to subsequent functions so they can put their windows and graphics on the virtual workstation screen.

Since these envelope routines work with either desk accessory programs or regular programs, the type of program must be distinguished in order to properly set up the desktop. Recall in the `config.c` file that `i_am_accessory` is set to 1 for a desk accessory and 0 for a regular program. If the program is a desk accessory, the window will be opened only if the user selects the name from the Desk menu; otherwise, the window should be opened immediately on running the program.

If this is a regular program, the GEM `wind_get` function is called to get the size of the desktop's work area. Normally `wind_get` is called with a window handle as its first argument. Here, however, the desktop is always window number 0.

The `setup_window` function is called to create a window that contains our work area. The `setup_window` function always creates a window that's ca-

pable of being full-size, but the window that appears first is the size dictated in the variables `fx`, `fy`, `fw`, and `fh`.

Next, the GEM functions `graf_mouse` and `Cursconf` are used to change the mouse to a pointing finger and to hide the TOS text cursor, which doesn't have any meaning in the GEM desktop environment.

Desk accessory programs need to behave differently: They must continuously run in the background, only appearing in a window when selected from the Desk menu. To make certain an accessory program never exits, these lines are added to stay in a loop:

```
do {  
    if( open_data(wh,vw_hand,file ))  
        multi (events,&wh,interval,wind_name,&vw_hand);  
    } while ( i_am_accessory );
```

In the loop are calls to the function `open_data`, which handles any data files for this program, and `multi`, which handles all input from the mouse or the keyboard, or messages from GEM that an environment event has occurred (such as a window being resized). If the `multi` subroutine returns, and `i_am_accessory` is true (1), `multi` will be called again.

The `do_cleanup` routine is called whenever there are some application-specific tasks that must be performed before the program can exit, such as saving a game score, closing files, and updating records.

```
do_cleanup(whand,vw)  
int whand, vw;
```

The code for `do_cleanup` supplied here doesn't do anything, since it's only useful for an individual program's cleanup requirements and must be tailored to that program. It's only included here as a dummy routine to give our library a complete set of routines.

For a regular program, don't loop, but rather call the `close_all` function to close the window, and the GEM `v_clsvwk` function to close the virtual workstation. Last, call GEM's `appl_exit` to exit our application program.

Program 2-2. `main.c`

```
/*  
** This is where we begin.  
** We set up the screen so we can write things on it.  
** We save the old color map so we can reset it before we exit.  
** We set up windows with sliders, a title, etc.  
** We change the mouse from an arrow to a pointing finger (for fun).  
** We call multi() to handle all of the mouse and keyboard input.  
** Then we restore the color map and exit.  
*/  
  
# include <osbind.h>  
# include <gemdefs.h>  
  
# define HIDE_CURSOR 0
```

CHAPTER 2

```
# define SHOW_CURSOR 1

main(ac,av)
int ac;
char **av;{

    extern struct object *main_addr;
    extern int i_am_accessory, interval, events;
    extern char *wind_name;
    char *file;
    int wh, vw_hand, fx, fy, fw, fh;

    vw_hand = setup_screen();
    wh = -1;
    if( i_am_accessory == 0 ){
        wind_get(0, WF_WORKXYWH, &fx, &fy, &fw, &fh );
        wh = setup_window(wind_name,0,0,1000,1000,fx,fy,fw,fh);
        graf_mouse( POINT_HAND, 0L );
        Cursconf(HIDE_CURSOR,0);
    }
    if( ac > 1 )
        file = av[1];
    else
        file = "";
    do {
        if( open_data(wh,vw_hand,file) )
            multi(events,&wh,interval,wind_name,&vw_hand);
        } while( i_am_accessory );
    do_cleanup(wh,vw_hand);
    close_all(main_addr,wh);
    v_clsvwk(vw_hand);
    appl_exit();
}
```

The setup_screen Function

To set up the screen, GEM must be informed that libraries are being used. Next the virtual workstation must be opened, which tells GEM the screen is being used, and the types of lines and colors to use. Program 2-3 is the program which does this.

The main routine calls the setup_screen function in order to open a window on the screen in which our C program can display its screen output.

The setup_screen function uses several GEM-supplied functions to obtain certain information about the screen. First, it calls appl_init to initialize GEM's internal state so that it's possible to call other GEM functions. Next, setup_screen calls graf_handle, which returns the width and height of a character in the font used in menus and dialog boxes, the width and height of a box large enough to hold a few characters in one of the screen "buttons," and an integer that represents a GEM Virtual Device Interface workstation handle. This integer is passed to our open_vwork function, which converts it into a virtual workstation handle the C program will use, such as for screen drawing and clearing operations. Then, setup_screen returns all this information to the main routine.

For a desk accessory program, setup_screen calls the GEM menu_register function, which enters the accessory's name (specified in the config.c file) into the Desk menu on the desktop.

A virtual workstation for an accessory program is opened only after the user has clicked on the accessory name in the Desk menu. (Regular applications open the virtual workstation immediately.) Because some routines need to know when the virtual workstation is open, `setup_screen` returns `-1` to main, where main puts it into the `vw_hand` variable to let other routines know that it's closed.

Program 2-3. `setscrn.c`

```
/*
** To set up the screen, we must inform GEM that we are using its
** libraries. Then we must open the "virtual workstation", which
** tells GEM that we are using the screen, and what types of lines
** and colors we want. To open the virtual workstation, we need to
** get a "handle" from graf_handle() which points to the screen.
*/

# include <gemdefs.h>

# define NO_VWS          -1

int gl_hchar, gl_wchar, gl_wbox, gl_hbox;    /* size of characters */
int menu_id;                                /* accessory handle */

setup_screen() {

    extern int i_am_accessory, gl_apid;
    extern char *access_name;
    int gr_handle;

    appl_init();
    gr_handle = graf_handle(&gl_wchar, &gl_hchar, &gl_wbox, &gl_hbox);
    if( i_am_accessory ) {
        menu_id = menu_register(gl_apid, access_name);
        return(NO_VWS);
    }
    return( open_vwork(gr_handle) );
}
```

The `open_vwork` Function

The `open_vwork` function, Program 2-4, tells GEM how to set up the virtual workstation: which colors to use, graphics characteristics, and device type.

Eleven input parameters are defined and stored in an array whose address is passed to `open_vwork`. A loop is used to set all the values to one, and then the first and last items are set to different values. The 11 parameters in the array are:

- 0 The device ID number. Use `V_SCREEN` for the screen. Printers, plotters, cameras, files, and other devices can be specified if they are supported. The constants defined at the beginning of the routine listing use the device ID numbers that are listed in the *Atari ST Software Developer's Kit* documentation.
- 1 Linetype 1 is a solid line
- 2 Poly Line color 1 is black
- 3 Poly Marker type 1 is a dot

CHAPTER 2

- 4 Poly Marker color 1 is black
- 5 Typeface 1 is the system type, as in menus
- 6 Text color 1 is black
- 7 Fill interior style 1 is hollow
- 8 Fill style 1 is an empty pattern (no pixels on)
- 9 Fill color 1 is black
- 10 This is the flag for Normalized Device Coordinates (NDC) or Raster Coordinates (RC). A value of 2 selects RC and a value of 1 selects NDC. We'll use RC, even though it requires that we know the exact screen size since the raster coordinates correspond to physical positions. Graphics operations are faster with RC because coordinates don't need converting for the output device. With NDC, coordinates are given in numbers between 0 and 32767 and produce a graphically correct image on any peripheral device. However, NDC takes longer because the coordinates must be converted to the appropriate raster coordinates.

The `open_vwork` function calls the GEM function `v_opnvwk`, which, besides putting the 11 parameter values into an array, converts the Virtual Device Interface handle returned by `graf_handle` in `setup_screen` to a GEM virtual workstation handle. This handle value is returned to main for use with subsequent subroutines. This GEM function also returns 58 values—in an array—that tell the height and width of the screen in pixels, the total number of line type choices, fill patterns, and so forth.

Program 2-4. `openvwrk.c`

```
# define V_SCREEN      1
# define V_PLOTTER     11
# define V_PRINTER     21
# define V_METAFILE    31
# define V_CAMERA      41
# define V_GRAFTAB     51

open_vwork(handle)
int handle;{

    register int i;
    static int in[11], out[57];

    for( i = 0; i < 10; i++ )
        in[i] = 1;
    in[0] = V_SCREEN;
    in[10] = 2;
    /*
    ** handle comes in a graf_handle, and goes out a vw_handle
    */
    v_opnvwk(in,&handle,out);
    return( handle );
}
```

The setup_window Function

Whenever a routine wants to open a window, complete with sliders and menus, it can call the setup_window function, Program 2-5. In the envelope library, main calls this function for regular programs and the was_msg function (discussed below) calls it for desk accessory programs.

The setup_window parameters are the window name and the size of the desktop work area and of the window slider boxes. For a desk accessory program, which doesn't use menus, these arguments are simply passed to open_window, which then opens the accessory window.

For regular programs, usually you'll want to add menus, messages, and dialog boxes to the program window. GEM needs to know the address of the text and graphic items. For menu structures and other items constructed with the Resource Construction Set from the *Atari ST Software Developer's Kit*, load the resource file for the menu tree structure and get the address of the root of the menu tree using the GEM functions rsrc_load and rsrc_gaddr. If the menus were constructed using the build_tree function (discussed below), then call build_tree, which returns the starting address of the tree.

The data file is read into an area of RAM allocated by GEM in the free memory left over beyond your program, and then the file is closed.

The address of the menu tree root is placed in the variable main_addr by rsrc_gaddr.

Note that you have the option of making build_tree return 0, in which case the program has no menus. This might be useful for very simple programs; however, the user probably should be given access to the Desk and File menus to start desk accessories and quit the program.

Program 2-5. setwind.c

```
/*
** There are two ways to handle menus.
** One way is to define the menu structure
** yourself in an array of object structures.
** The other way is to construct a resource file
** with the Resource Construction Set.
** Using RCS is easy, defining your own structures
** can be tedious. However, defining the structures
** in your own program means that you only need one file
** instead of two, and it has the advantage of being
** printable.
*/

#include <gemdefs.h>
#include <obdefs.h>

struct object #main_addr = 0;

#include <window.h>

setup_window(name, vp, hp, vs, hs, dx, dy, dw, dh)
char #name;
int vp, hp, vs, hs, dx, dy, dw, dh;

extern char #resource;
```

```

extern int i_am_accessory;
struct object *build_tree();

if( i_am_accessory == 0 ){
    if( resource ){
        rsrc_load( resource );
        rsrc_gaddr( R_TREE, MAINMENU, &main_addr );
    }

    else {
        main_addr = build_tree();
    }

    if( main_addr ){
        menu_bar( main_addr, 1 );
    }
}

return( open_window( name, vp, hp, vs, hs, dx, dy, dw, dh ) );
}

```

The open_window Function

The open_window function, Program 2-6, is called by setup_window to create an open window on the screen by expanding a box outline to the correct window size, placing horizontal and vertical slider boxes in position, and returning the initial values.

Program 2-6. openwind.c

```

/*
** Create the window, grow a box for effect, and open the window.
** Arrange for horizontal and vertical sliders
** to exist, and set them to initial values passed to us.
*/

# include <gemdefs.h>
# include <osbind.h>
# include <wfparts.h>

open_window(name,vertical,horizontal,vsize,hsize,dx,dy,dw,dh)
char *name;
int vertical, horizontal, vsize, hsize, dx, dy, dw, dh;{

    int wi_handle, fx, fy, fw, fh;

    wind_get(0, WF_WORKXYWH, &fx, &fy, &fw, &fh );
    wi_handle = wind_create(WF_PARTS,fx,fy,fw,fh);
    wind_set(wi_handle, WF_NAME,name,0,0);
    wind_set(wi_handle, WF_VSLSIZE,vsize,0,0);
    wind_set(wi_handle, WF_HSLSIZE,hsize,0,0);
    wind_set(wi_handle, WF_VSLIDE,1000-vertical,0,0,0,0);
    wind_set(wi_handle, WF_HSLIDE,horizontal,0,0);
    graf_growbox( dx + dw/2, dy + dh/2, 2, 2, dx, dy, dw, dh );
    wind_open(wi_handle,dx,dy,dw,dh);
    return( wi_handle );
}

```

The arguments used by open_window are the window name, window size, and the size of the slider boxes.

First, open_window calls GEM's wind_get function to get the size of the desktop's work area. This will be the maximum size of the window that is created. Note that a window is created with the potential to be full-size, even for a

desk-accessory type of program which usually opens in a small window. This is done so a user can resize a small window to the full desktop size.

`wind_create` is called to set up the data structure for the window and store the window handle returned by this function so this window can be referred to later. This window handle is distinct from the virtual workstation handle, which is used for graphics operations. The handle created by `wind_create` is used by the GEM `wind_get` and `wind_set` functions to control the window.

The `wind_set` routine tells GEM how long to make the slider boxes and where to place them in the window that's been created.

To display an expanding box that simulates an enlarging window when the window is opened, we call the `graf_growbox` function.

Finally, GEM opens a window on the screen with the `wind_open` function that corresponds to the size passed to it by `open_window`, which has had its arguments passed from the `setup_window` subroutine.

The window size information used by `wind_open` and `graf_growbox` may be different depending on whether this is a regular program or an accessory. For a regular program, the window opens to the full desktop area. For a desk accessory, which usually appears in a smaller window, the window size specified in the `config.c` file is used. Although a desk-accessory window usually is small when it opens, the user can still expand it to fill the desktop because the window was created full size earlier in this subroutine.

The window handle created by `wind_create` is returned to `setup_window`, which returns it to main.

The `open_data` Function

Programs frequently need to open data files, initialize data, ask for input, and perform other similar operations while they are executing. With the `open_data` function, Program 2-7, a C program using the GEM interface can communicate with TOS or call other GEM routines.

If your C program requires a data file, you can enter the argument for the data file in the program's `open_data` subroutine. An example of this can be found in the PLOT program which appears later in this book. This routine serves to "connect" the C application program to GEM and TOS.

The `open_data` routine is called from main and is passed an argument that is either a null string or the string that is entered as a parameter on the command line when the program is run. You can make your program be a TOS-Takes-Parameters program by adding the `.TTP` extension to its name; then your application will accept filenames from the command line. For example, with the program PLOT.TTP, you could type on the command line:

plot expenses.dat

and the plot program would know that the file named `expenses.dat` holds its data values.

If your application program requires this function, you have to create it for the specific application. Later, when the PLOT program is discussed, we'll explain how to write this routine.

For our purposes of having a prebuilt library of functions that generate a general-purpose GEM interface, we've included a version that does nothing except return a value of 1 to indicate the routine was successful. Zero is returned if it was not.

This default version of `open_data` makes it convenient to compile and link application programs that don't use the function. If you do not supply your own version, the linker will find this default version in the library, and use it. If you do have your own version, then the linker will know not to look for another copy in the library. This way, the linker never has to be changed to work with applications that don't use this function.

Program 2-7. `opendata.c`

```
open_data(whand,vw,file)
int whand, vw;
char *file;{
```

```
    return(1);
}
```

The multi Function

Application programs generally require some input—such as a mouse click, keypress, or mouse movement—while running. After finishing with the `open_data` function instructions, the main routine calls the multi function, Program 2-8, to handle all input. In main, the statement

```
multi(events,&wh,interval,wind_name,&vw_hand)
```

passes the multi event variables defined in the `config.c` file to specify the input types multi should respond to. Our file lists messages from GEM (`MU_MESAG`), mouse clicks (`MU_BUTTON`), keypresses (`MU_KEYBD`), and special rectangles on the screen (`MU_M1` and `MU_M2`) as types of input multi should recognize.

Also passed to multi are addresses of the handles for the window and the virtual workstation so that functions called by multi can change these values as new windows or virtual workstations are created. The *address* of the value for the handle, rather than a copy of the value, is passed because you can't change the handle if you don't know its memory location. The other two arguments for the multi function are the variables from the `config.c` file that define the window name and the timer interval.

The multi function is essentially a loop that calls the GEM function `evnt_multi`, which monitors the system waiting for an input signal. When an input event occurs, `evnt_multi` passes the information to multi, which determines the proper function to call for the input type.

First, multi considers the interval period that originated in the config.c file and is now stored in the variable milli_secs, and if it isn't 0, the GEM MU_TIMER routine is added to the list of events to monitor. Fine and coarse clock settings are created as timer_high and timer_low, to be used by evnt_multi in waiting for input events. Note that our routines use only one 16-bit word for the interval, so intervals can be a maximum of only 65 seconds.

This is generally enough time for most waiting periods, but if you want a longer interval, you can change the code to make the interval and milli_secs variables 32-bit numbers.

The multi function passes to evnt_multi a long list of parameters:

1. Which events to wait for: those listed in the config.c file and the timing interval (if it isn't 0).
2. How many mouse clicks to wait for before responding. Two allows double clicks to be recognized. Note that one mouse click consists of a button-down and a button-up signal.
3. Which mouse buttons to respond to: 1 is the left button, 2 is the right button, and 3 is both buttons.
4. What button action to wait for (up or down). Since we usually want to know about any mouse button activity, we define the btn variable to be the opposite of the current button state. If the button is down, btn is set to "wait for button to come up." If the button is up, btn is set to "wait for the button to go down."
5. Two sets of mouse rectangles are definable. A mouse rectangle is an area on the screen that sends an event signal when the mouse cursor enters the area—for example, when the cursor points to a command. Since mouse rectangles are not used, zeros are placed here. If valid values x, y, w, and h were placed here, GEM would send a message each time the mouse cursor entered the rectangle.
6. m is a variable in an array of 8, where interprocess communication (messages from GEM) and timer messages are stored. These are GEM messages communicating that a window has been either moved, resized, or closed, or that a slider box has been moved, and so on.
7. The timer_low and timer_high variables.
8. The addresses of the variables that will hold the x and y coordinate values for the mouse after an event has occurred.
9. The state of the mouse buttons after the latest mouse event.
10. The state of the shift keys.
11. The character that was typed for a keyboard event.
12. The number of mouse clicks.

After an event has occurred and evnt_multi has returned to multi, the mouse is displayed with

show_mouse()

just in case multi has been called from somewhere other than the main routine, and the mouse is hidden.

Then multi decides what further action to take, based on which event has occurred. The event may be a GEM message. The messages that multi receives from GEM consist of eight integers in the array m[8].

m[0] is the message type
 m[1] is the application ID of the sender
 m[2] if nonzero, then there is more data beyond the eight integers. With GEM itself there never is, and this integer is always zero.

The remaining integers, m[3] to m[7], vary depending on the value in m[0]. If m[0] is WM_REDRAW, WM_SIZED, or WM_MOVED:

m[3] is the window handle
 m[4]-m[7] are the x, y, w, h values for the operation

If m[0] is WM_TOPPED, WM_CLOSED, WM_FULLED, or WM_NEWTOP, then

m[3] is the window handle
 m[4]-m[7] are ignored

If m[0] is AC_OPEN, then

m[3] is ignored
 m[4] is the menu ID
 m[5]-m[7] are ignored

If m[0] is AC_CLOSE, then

m[3] is the menu ID
 m[4]-m[7] are ignored

If m[0] is WM_VSLID or WM_HLID, then

m[3] is the window handle
 m[4] is the slider box position
 m[5]-m[7] are ignored

If m[0] is WM_ARROWED, then

m[3] is the window handle
 m[4] is the arrow that was clicked: page-up, page-down, and so on

If m[0] is MN_SELECTED, then

m[3] is the menu title
 m[4] is the menu item
 m[5]-m[7] are ignored

These last elements tell the program which menu tree has been chosen and the index in the tree of the menu item.

If the event is one of the messages from GEM explained above, the

was_msg function is called and the relevant information about the message is passed. The application ID and message length are not needed (the second and third message fields, m[1] and m[2]), so they're omitted from the argument list.

If the event is a keystroke, multi calls gotkey. You define the gotkey function for a specific program so that it returns a nonzero value if the program should exit when a key is pressed. The gotkey version (shown later) provided with the envelope routines returns the value 1. If a program doesn't define how to handle keystrokes, it will exit when a key is pressed.

For a mouse button click, our mouse_hit function is called. This function returns the new state of the mouse button—whether it's up or down—so that evnt_multi will be watching for the opposite button state.

Lastly, if evnt_multi detects an event that doesn't correspond to any of the events we've described, the program notifies us with "What?" in a window created by the show_form function. A user should never see this message, but during the development phases it will alert you (the programmer) that something is amiss and unplanned events are occurring.

Program 2-8. multi.c

```

/*
** This is where all input is handled.
** We worry about timer messages, mouse buttons, keyboard typing,
** and messages about window activity.
** We loop, calling evnt_multi(), until we get a message saying
** that the user clicked on the CLOSE patch in the upper left
** corner of the window.
*/

#include <gemdefs.h>
#include <osbind.h>

multi(events,wh,milli_secs,name,vw)
int events;
int *wh;
int milli_secs;
char *name;
int *vw;{

    static int event, timer_low, timer_high, mu_timer, m[8];
    static int bbutton, kstate, nclick, mx, my, keycode, btn, r;

    btn = 1;
    for(;;){
        timer_high = 0;
        if( milli_secs ){
            timer_low = milli_secs;
            mu_timer = MU_TIMER;
        }
        else {
            timer_low = 0;
            mu_timer = 0;
        }
        event = evnt_multi(events ! mu_timer,
            2,                /* how many clicks possible */
            3,                /* any buttons can click */
            btn,              /* if button down, wait for up, etc. */
            0,0,0,0,0,        /* mouse rectangle 1 */
            0,0,0,0,0,        /* mouse rectangle 2 */

```

```

m,                /* ipc & timer messages */
timer_low,        /* low word of timer value */
timer_high,       /* high word */
&mx,&my,          /* mouse coordinates */
&bbutton,         /* mouse button states */
&kstate,          /* shift key states */
&keycode,         /* the key that was hit */
&nclick /* number of mouse clicks hit */
);

show_mouse();
if( event & MU_MESAG ){
    if( r=was_msg(m[0],m[3],m[4],m[5],m[6],m[7],wh,name,vw) )
        return(r);
}
else if( event & MU_KEYBD ){
    if( r = got_key(keycode,&wh,&vw) )
        return(r);
}
else if( event & MU_BUTTON ){
    btn = mouse_hit(btn,mx,my,kstate,nclick,&wh,&vw);
}
else if( event & MU_TIMER ){
    clock_ticks(&wh,&vw);
}
else {
    show_form("What?");
    do_display(&wh,&vw);
}
}
}

```

The show_form Function

The more messages we can provide for ourselves while debugging a program, the better. By using the show_form function, Program 2-9, we can put a message in its own window when the screen may otherwise be busy or unavailable. It prints a string in the window along with an OK button and a CANCEL button.

The show_form function calls GEM's sprintf function to put our string into the form needed by the GEM form_alert function. The form_alert function takes the string and a variable for the button that we want to be the default as its arguments and returns the number of the button that was clicked. In this program, the OK button returns 1 and the CANCEL button returns 2. The button value is converted to true/false by subtracting 1. Therefore, an if statement can be used to decide what action to take, depending on whether show_form returns true or false.

Program 2-9. showform.c

```

/*
** Here is a small routine that shows some information in a window
** and waits for the user to read it.
*/

show_form(s)
char *s;{

    char str2[128];

```



```
sprintf(str2, "[1][%s][ OK : CANCEL ]", s);  
return( form_alert(2, str2) - 1 );  
}
```

The was_msg Function

The was_msg function, Program 2-10, is the heart of the envelope library. It takes care of messages received by the GEM evnt_multi function and performs the window management routines that characterize GEM programs.

In the following descriptions of the was_msg processes, numerous GEM-defined constants are used as arguments. The discussion doesn't go into much detail about them except to explain their relationships to the functions in which they are used. For a more detailed discussion see the developer's documentation from Atari.

When the multi function calls was_msg, it passes all the information about the event that occurred in a form that is ready to be processed. The processing in was_msg is basically one switch statement in which the information in the msg parameter determines the appropriate action.

GEM's wind_update routine is called with a 1 parameter to insure that the window updates are not confused with window updates from other programs. The wind_update function locks out other window activity, from desk accessories or GEM itself, until we're finished. At the conclusion of the switch statement, wind_update is called again, this time with a 0 parameter, to unlock the window updating. If the window updating isn't unlocked, the system cannot change windows and will appear to be hung up, requiring a system reboot. Note that for each case in the switch statement, the last action is to break out of the switch to insure that wind_update is turned off.

If the user has clicked on the close box in the upper left corner of the window, then evnt_multi returns the GEM message WM_CLOSED, which is then passed to the current was_msg function. In

case WM_CLOSED:

the window is closed, the window handle is set to NO_WINDOW to prevent its further use, and the exit flag is set to the value BYE_BYE, recognized by multi as meaning exit. (Accessory programs don't really exit—main will refuse to exit and will call multi again.)

When the user selects an item from the menu, GEM sends the message MN_SELECTED to evnt_multi. In

case MN_SELECTED:

the application-specific do_menu function is called with the menu number and the item within the menu. Later, we'll discuss how to write the do_menu function for a particular application. do_menu also determines whether or not the menu item selected means to exit the program, like Quit in the File menu. If do_menu returns a nonzero value, meaning to exit, then the exit flag is set and we break out of the switch.

If some other screen activity causes the program's window to be overwritten, GEM returns the message `WM_REDRAW`. Then the program must redraw the part that was changed.

case `WM_REDRAW`:

calls the function `do_redraw`, which performs the complicated task of cleaning up the screen after a window has been moved, resized, or closed. As usual, we break out of this switch when done.

The next switch case,

case `AC_CLOSE`:

will be used only if the program is a desk accessory. When a user starts a program, GEM automatically closes the desk-accessory window (although the accessory program still runs) to free up the window. Eight is the maximum number of windows GEM allows and there's no point in having one used up when it isn't visible. When the current program ends, or the screen is about to be cleared, or GEM is re-initializing the window library data structures, GEM sends the message `AC_CLOSE` to `evnt_multi`. If the title that is passed matches the `menu_id` variable, then it knows that the window that was closed belonged to the accessory program. Once this information is known, the decision to reopen the accessory window—by calling the `setup_window` subroutine—or to take some other action can be made.

In our version of the `was_msg` function, the accessory window is left closed. Both the window handle and the virtual workstation handle are closed so the programs in the library don't try to use the nonexistent window. A special code is returned so that the routine that originally called `multi` can take application-specific action on `AC_CLOSE` if required.

When a user selects our accessory program from the Desk menu, GEM sends the `AC_OPEN` message. In

case `AC_OPEN`:

the title variable is set to the window handle for a reason that is explained below. First verify that the `menu_id` matches, then check that a virtual workstation and window are not already open, and finally open a virtual workstation and set up a window just as main does for regular programs. Last, instead of breaking out of the switch, the execution continues into the code which will make our window the topmost (active) window: `WM_TOPPED` and `WM_NEWTOP`. The reason the title variable is set to the window handle is so it can be used in `WM_TOPPED`.

GEM delivers the `WM_TOPPED` and `WM_NEWTOP` messages when the program window has been made the topmost one either by clicking on it, or by removing another window from the top:

case `WM_NEWTOP`:

case `WM_TOPPED`:

The `wind_set` is called with the value `WF_TOP` to make the window look like a top window by filling in its borders. The function `do_display` is called to cause the application program to display the window contents.

Moving or resizing a window causes GEM to issue the `WM_SIZED` and `WM_MOVED` messages. The code for

```
case WM_SIZED:
case WM_MOVED:
```

first constrains the window to the minimum size so the borders remain large enough to use. Then, the current full window is set to the size passed in the variables `x`, `y`, `w`, and `h`. The GEM `WF_CURRXYWH` function sets the values for the current window, including borders. Using the new size, the actual size of the work area is calculated and `do_display` is called to have the program add the contents of the window. GEM's `WF_WORKXYWH` function sets the values for the current window, excluding the borders.

When a user clicks in either the shaded area of the sliders or on the arrow icons at either end of the icons, GEM delivers the `WM_ARROWED` message. The switch code in

```
case WM_ARROWED
```

simply calls the `do_arrows` routine to take the proper action, and then `do_display` is called to redraw the screen.

If the GEM message is `WM_VSLID` or `WM_HSLID`, the user has moved one of the slider boxes. In

```
case WM_VSLID
```

and

```
case WM_HSLID
```

we call GEM routine `wind_set` with the window handle (stored in the title variable), the GEM-defined constant `WF_VSLIDE` or `WF_HSLIDE`, and the integer which represents where in the slider area the user has positioned the slider box. The slider box is set to the new value, and the application-specific functions, `v_touched` and `h_touched`, are called to make a particular program behave appropriately for the slider action—for example, scrolling text in an editor. `do_display` then redraws the screen.

When the user wants to expand a window to its largest size, or to return it to its previous size if it's already full-sized, he or she clicks on the icon in the upper right corner of the window. This causes GEM to send the message `WM_FULLED`. In the switch code for

```
case WM_FULLED
```

the GEM function `wind_get` returns the current window size using the GEM-defined constant `WF_CURRXYWH` as the parameter for the current window.

Likewise, by using the GEM constant `WF_WORKXYWH` as a parameter, you get the size of the desktop's work area. These two sizes are compared, and if they are equal, the program window is already full-size. To return the window to its previous size and location, `wind_get` is called again with the GEM constant `WF_PREVXYWH` as a parameter and the returned values are put into the `x`, `y`, `w`, and `h` variables. If the two windows are not equal size, simply set the current window to the variables `x`, `y`, `w`, and `h` (which were set to be equal to the size of the desktop's work area when we compared them), recalculate the size of the work area, and then set the size of the work area. GEM will send a `WM_REDRAW` message if the new window needs to be redrawn.

Program 2-10. `wasmsg.c`

```
/*
** Here we handle messages received by evnt_multi().
** If we were asked to close the main window, then we
** return non-zero, which will eventually cause us to exit.
*/

#include <gemdefs.h>
#include <osbind.h>
#include <wfparts.h>

#define MIN_WIDTH      10
#define MIN_HEIGHT     10
#define NO_WINDOW      -1
#define NO_VWS         -1
#define BYE_BYE        -1
#define OBLIVION        -2

was_msg(msg,title,x,y,w,h,whand,name,vw)
int msg, title, x, y, w, h, $whand;
char $name;
int $vw;{

    int exit_flag;          /* 0 = continue, BYE_BYE = exit */
    int xc, yc, wc, hc, j;
    extern int menu_id, slv, slh, svl, shs, sx, sy, sw, sh;

    exit_flag = 0;
    wind_update(1);
    switch( msg ){
        case WM_CLOSED:
            close_window( title );
            $whand = NO_WINDOW;
            exit_flag = BYE_BYE;
            break;
        case MN_SELECTED:
            if( do_menu(title,x,$whand,$vw) )
                exit_flag = BYE_BYE;
            break;
        case WM_REDRAW:
            do_redraw(x,y,w,h,title,$vw);
            break;
        case AC_CLOSE:
            if( title == menu_id ){
                $whand = NO_WINDOW;
                if( $vw != NO_VWS )
                    v_clsvwk( $vw );
                $vw = NO_VWS;
                exit_flag = OBLIVION;
            }
    }
}
```

```

    }
    break;
case AC_OPEN:
    title = $whand;
    if( x != menu_id )
        break;
    if( $vw == NO_VWS )
        $vw = open_vwork( graf_handle(&j,&j,&j,&j) );
    if( $whand == NO_WINDOW ) {
        title = $whand = setup_window(name,
            slv,slh,svs,shs,sx,sy,sw,sh);
        break;
    }
    /* Fall through to top the window */
case WM_NEWTOP:
case WM_TOPPED:
    wind_set(title,WF_TOP,0,0,0,0);
    do_display(title,$vw);
    break;
case WM_SIZED:
case WM_MOVED:
    if( w < MIN_WIDTH )
        w = MIN_WIDTH;
    if( h < MIN_HEIGHT )
        h = MIN_HEIGHT;
    wind_set(title,WF_CURRXYWH,x,y,w,h);
    wind_calc(WC_WORK,WF_PARTS,x,y,w,h,&x,&y,&w,&h);
    wind_set(title,WF_WORKXYWH,x,y,w,h);
    do_display(title,$vw);
    break;
case WM_ARROWED:
    do_arrows(x,title,$vw);
    do_display(title,$vw);
    break;
case WM_VSLID:
    wind_set(title,WF_VSLIDE,x,0,0,0);
    v_touched(title,$vw,x);
    do_display(title,$vw);
    break;
case WM_HSLID:
    wind_set(title,WF_HSLIDE,x,0,0,0);
    h_touched(title,$vw,x);
    do_display(title,$vw);
    break;
case WM_FULLED:
    wind_get(title,WF_CURRXYWH,&xc,&yc,&wc,&hc);
    wind_get(0,WF_WORKXYWH,&x,&y,&w,&h);
    if( wc == w && hc == h )
        wind_get(title,WF_PREVXYWH,&x,&y,&w,&h);
    wind_set(title,WF_CURRXYWH,x,y,w,h);
    wind_calc(WC_WORK,WF_PARTS,x,y,w,h,&x,&y,&w,&h);
    wind_set(title,WF_WORKXYWH,x,y,w,h);
    break;
}
wind_update( 0 );
return( exit_flag );
}

```

The do_menu Function

This function, Program 2-11, must be written for a specific application program. The envelope library contains the following default version, which simply returns a value of 0 to indicate that the user did not select "exit" or "quit." Later, you'll see how to develop this function for programs that need menus.

Program 2-11. domenu.c

```
do_menu(title,item)
int title,item;{

    return( 0 );
}
```

The do_redraw and just_draw Functions

Because GEM allows multiple overlapping windows, it can be a fairly complex task to redraw a program application's window when it is all or partly obscured by other windows. To handle this situation, GEM creates a list of subwindows, each of which is a rectangle.

do_redraw, Program 2-12, figures out what part of the application window is newly exposed after a covering window has been moved away. Then it sets a clipping window to the size and location of the part that will be redrawn, so that only the points within the rectangle will appear on the screen. All others are "clipped off" and not drawn.

At the start of this function, the mouse is hidden to prevent its being drawn over. If it isn't hidden, it will leave traces of the old window behind it when it is moved.

Another important precaution is to set the wind_update function to 1, to lock out all other processes until this one is finished. The window will be unlocked before exiting this routine.

In the do_redraw code, the declaration

```
GRECT t1, t2;
```

specifies a particular data structure for rectangles. This structure is used later in do_redraw by the GEM rc_intersect routine. The rectangle represented by t2 is the part of our application window that was covered by the window that's now gone. GEM has told us about the intersecting area of our window and the covering window. During the redrawing process, GEM subdivides the window into subrectangles.

The t2 rectangle is then compared to each of the rectangles in GEM's list to see if they intersect. When a t1 rectangle intersects a t2 rectangle, just_draw, Program 2-13, is called to redraw the part of the screen defined by the t1 rectangle. rc_intersect changes the t1 rectangle to be the intersection of the old t1 and t2.

Program 2-12. redraw.c

```
/*
** This routine worries a lot about how to clean up the screen after
** a window has re-sized, moved, or disappeared. Since there can be
** many overlapping windows, the task is not trivial. The trick used
** is to define a list of rectangles formed wherever the window is
** visible, and then to refresh each rectangle (using the clipping
** functions) wherever it overlaps the dirty rectangle passed to us.
** rc_intersect returns TRUE if there is an overlap, and it puts the
** overlap into its second argument.
```



```

** The function wind_get() is used to get the FIRST rectangle in the
** list, then used again in a loop to get each NEXT rectangle.
*/

# include <obdefs.h>
# include <gemdefs.h>

do_redraw(xc,yc,wc,hc,whand,vw)
int xc, yc, wc, hc, whand, vw;{

    int clip[4];
    GRECT t1, t2;

    hide_mouse();
    wind_update(1);
    t2.g_x = xc;
    t2.g_y = yc;
    t2.g_w = wc;
    t2.g_h = hc;
    wind_get(whand,WF_FIRSTXYWH,&t1.g_x,&t1.g_y,&t1.g_w,&t1.g_h);
    while (t1.g_w && t1.g_h) {
        if (rc_intersect(&t2,&t1)) {
            clip[0] = t1.g_x;
            clip[1] = t1.g_y;
            clip[2] = t1.g_x + t1.g_w - 1;
            clip[3] = t1.g_y + t1.g_h - 1;
            vs_clip(vw,1,clip);
            just_draw(whand,t1.g_x,t1.g_y,t1.g_w,t1.g_h,vw);
        }
        wind_get(whand,WF_NEXTXYWH,&t1.g_x,&t1.g_y,&t1.g_w,&t1.g_h);
    }
    wind_update(0);
    show_mouse();
}

```

Program 2-13. justdraw.c

```

just_draw(whand,x,y,w,h,vw)
int whand, x, y, w, h, vw;{

    do_display(whand,vw);
}

```

The do_arrows Function

This function, Program 2-14, isn't limited to arrows, as its name implies. It handles the positioning of the slider boxes and screen scrolling that occurs when a user clicks on the arrow icons at either end of the slider areas, or clicks on the gray portion of the slider area.

In most programs, the screen contents appear to move up or down, right or left when the user clicks in the slider area. Clicks on the arrow icons in the vertical slider area cause the window to move one row up or one row down. In the horizontal slider area, the arrows move the window one column to the right or left. In the same way, clicks in the gray areas of the vertical and horizontal slider areas move the window one page up or down and one page right or left.

This `do_arrows` function calculates the values for window movements and slider box positioning.

Using GEM's `wind_get` function to find out the current size of the work area, `do_arrows` calculates just how many lines and columns of text fit in the current window.

A GEM message is passed from the `was_msg` function to the `do_arrows` function, declaring that there's been an event in the slider area such as `PAG_UP`, `ROW_DN`, and `PAG_RT`. You can see these messages used in the switch statement in this function.

Depending on the event, the current line and column positions are calculated in the switch statement. Line and column operations only require incrementing or decrementing the `cur_line` and `cur_col` variables by 1, whereas page operations require using the numbers that were calculated for the current window size.

After setting `cur_line` and `cur_col` to their new values, `do_arrows` calls our function `slide_pos` to calculate the new positions of the slider boxes, and then calls GEM's `wind_set` subroutine to set the sliders.

Program 2-14. `doarrows.c`

```
# include <document.h>
# include <gemdefs.h>

# define PAG_UP          0
# define PAG_DN          1
# define ROW_UP          2
# define ROW_DN          3
# define PAG_LF          4
# define PAG_RT          5
# define COL_LF          6
# define COL_RT          7

int xlines;
int cur_line, cur_col;

do_arrows(operation,whand,vw)
int operation, whand, vw;{

    int x, y, w, h, wlines, wcols;
    extern cur_line, cur_col, gl_wchar, gl_hchar, xlines;
    int vertical, horizontal;

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    wlines = h / gl_hchar;
    wcols = w / gl_wchar;
    switch(operation){
        case PAG_UP:
            cur_line -= wlines;
            if( cur_line < 0 )
                cur_line = 0;
            break;
        case PAG_DN:
            cur_line += wlines;
            if( cur_line > NLINES - wlines )
                cur_line = NLINES - wlines;
            break;
        case ROW_UP:
            cur_line--;
```

```

        if( cur_line < 0 )
            cur_line = 0;
        break;
    case ROW_DN:
        cur_line++;
        if( cur_line > NLINES - wlines )
            cur_line = NLINES - wlines;
        break;
    case PAG_LF:
        cur_col -= wcols;
        if( cur_col < 0 )
            cur_col = 0;
        break;
    case PAG_RT:
        cur_col += wcols;
        if( cur_col > NCHARS - wcols )
            cur_col = NCHARS - wcols;
        break;
    case COL_LF:
        cur_col--;
        if( cur_col < 0 )
            cur_col = 0;
        break;
    case COL_RT:
        cur_col++;
        if( cur_col > NCHARS - wcols )
            cur_col = NCHARS - wcols;
        break;
    }
    slide_pos( wlines, xlines, cur_line, &vertical );
    slide_pos( wcols, NCHARS, cur_col, &horizontal );
    wind_set(whand, WF_VSLIDE, vertical, 0, 0, 0);
    wind_set(whand, WF_HSLIDE, horizontal, 0, 0, 0);
}

```

The slide_pos Function

One of the particularly nice features about GEM's sliders is that the slider box is proportional to the total size of the document being scrolled. Placing the box requires that both the size and position for each slider box be computed. Both the slider area and the box range from 1 to 1000 in size. The slider box size is computed in the slide_size function. The position of the slider is calculated in slide_pos.

In slide_pos, Program 2-15, the function takes the line number of the document and calculates the position of the top of the slider box in the range of 1 to 1000. Keep in mind that the range is always measured to the top of the vertical slider box and to the left edge of the horizontal slider box. It is this part of the box that is positioned within the range.

To calculate the actual position of the top (or left side) of the slider box, the function multiplies the line number by the maximum slider position of 1000 and divides by the total document size minus the part that's visible. The result is returned to the GEM wind_set routine to plot the slider box in its new position, reflecting the position of the document fraction in the window with respect to the total document.

Program 2-15. slidepos.c

```

/*
**      Map a line number into a slider position between
**      0 and ( 1000 - the width of the slider )
**      Line numbers are numbered from 0 to nlines-1.
**      Works for columns also.
*/
slide_pos( visible, total, line, pos )
int visible, total, line, *pos;{

    *pos = 1000L * line / (total - visible);
}

```

The h_touched, v_touched, and pos_slide Functions

Another method by which the user scrolls the contents of a program window is “grabbing” the slider box with the mouse and sliding it in the slider area. For your program, the problem becomes how to tell which lines of the document correspond to the position of the top of the slider box (or the left edge of the horizontal slider).

To solve the problem, the functions `h_touched` and `v_touched` (Programs 2-16 and 2-17) are called to alter the global variables `cur_line` and `cur_col` whenever the user moves the slider box. Then `cur_line` and `cur_col` are used by application programs to display the relevant part of scrollable objects, such as a document.

To determine the new line or column positions from the slider box’s position in the possible range of 1–1000, `pos_slide` function is called. This function reverses the operation completed earlier in the `slide_pos` function (Program 2-15). It calculates the new line or column number based on the number of lines or columns in the document and the window, and the maximum position of the top (or left) of the slider box.

Program 2-16. htouched.c

```

# include <gemdefs.h>
# include <document.h>

h_touched(whand,vw,horizontal)
int whand, vw, horizontal;{

    int x, y, w, h, wcols;
    extern int gl_wchar, cur_col;

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    wcols = w / gl_wchar;
    pos_slide( wcols, NCHARS, &cur_col, horizontal );
}

```

Program 2-17. vtouched.c

```
# include <gemdefs.h>

v_touched(whand,vw,vertical)
int whand, vw, vertical;{

    int x, y, w, h, wlines;
    extern int gl_hchar, cur_line, xlines;

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    wlines = h / gl_hchar;
    pos_slide( wlines, xlines, &cur_line, vertical );
}
```

Program 2-18. posslide.c

```
/*
**      Map a slider position into a line number (or column)
**/
pos_slide( visible, total, line, pos )
int visible, total, *line, pos;{

    *line = (pos * (total - visible) ) / 1000L;
}
```

The hide_mouse and show_mouse Functions

As a programming convenience—so we don't have to worry about how many times the mouse has been hidden or exposed—we keep the `hide_mouse` and `show_mouse` functions in our envelope library (Program 2-19).

Whenever you want to be sure the mouse is either hidden or displayed, call one of these functions. These two functions track the status of the mouse by setting the `mouse_gone` variable to the current mouse condition. When either of the routines is called, it first checks this variable to determine whether the current mouse state is the one called for the program. If it is, the routine does nothing and returns.

Program 2-19. hidemous.c

```
/*
** These routines keep us from having to worry about how many
** times we hid the mouse, and how many times we tried to show
** it.
**/

# include <gemdefs.h>

static int mouse_gone;          /* is mouse visible? */

hide_mouse(){

    if(! mouse_gone){
        graf_mouse(M_OFF,0x0L);
        mouse_gone = ! mouse_gone;
    }
}
```

```
show_mouse() {
    if (mouse_gone) {
        graf_mouse (POINT_HAND, 0x0L);
        graf_mouse (M_ON, 0x0L);
        mouse_gone = ! mouse_gone;
    }
}
```

The do_display and doit Function

These two functions are used in conjunction with one another. The small do_display function, Program 2-20, makes sure the mouse is hidden before any screen redrawing occurs with the doit function. If the mouse is showing when the screen is redrawn, then a fragment of the earlier screen will be seen when the mouse is moved.

As you can see, do_display begins with a call to hide the mouse; then it calls the application-specific doit routine with the window handle and the virtual workstation handle, and redisplay the mouse when doit is finished.

The doit function, Program 2-21, is responsible for drawing the screen. It must be defined for each program's screen output needs. For many applications, especially those that only return information like maps and pictures, this is where most of the program's work occurs. In our envelope library, we include the following default version, which only clears the screen.

Elsewhere in this book, there are several examples of more complicated versions of this function.

Program 2-20. dodisp.c

```
/*
** Call the user's paint-screen routine 'doit()' to put
** something on the screen. We hide the mouse while it does it
** so that the mouse won't leave a stain.
*/
```

```
do_display (whand, vw)
int whand, vw; {
```

```
    hide_mouse();
    doit (whand, vw);
    show_mouse();
}
```

Program 2-21. doit.c

```
doit (whand, vw)
int whand, vw; {
```

```
    just_clear (whand, vw);
}
```


The just_clear Function

The purpose of just_clear, Program 2-22, is to clear the work area by drawing a white bar the width and length of the work area or a subwindow rectangle used by just_draw.

First, as a precaution in case just_clear has been called by a routine that didn't hide the mouse, just_clear calls the hide_mouse function.

Then some GEM routines are called to set the interior fill style to 2 (for pattern), the fill style index to 8 (for solid color), and the fill color to white. wind_get is used to get the size of the work area and to call the GEM v_bar routine to generate a rectangle that fills our program's work area with white. Before exiting just_clear, the fill color is set back to black (1) to show the mouse.

Program 2-22. justclr.c

```
/*
** Clear the display by drawing a white bar whose width is the screen
** width and whose length is the screen length.
*/
```

```
# include <gemdefs.h>
# include <obdefs.h>
```

```
just_clear(whand,vw_handle)
int whand, vw_handle;{
```

```
    int temp[4];
    int x, y, w, h;
```

```
    hide_mouse();
    vsf_interior( vw_handle, 2 );
    vsf_style( vw_handle, 8 );
    vsf_color( vw_handle, WHITE );
    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    temp[0] = x;
    temp[1] = y;
    temp[2] = x + w - 1;
    temp[3] = y + h - 1;
    v_bar( vw_handle, temp );
    vsf_color( vw_handle, 1 );
    show_mouse();
}
```

The clr_display and clip_work Functions

These functions, Programs 2-23 and 2-24, are used when you know you want to clear the entire work area of a program window. They reset the clipping window to be the size of the work area. You would not call these functions from just_draw, which needs to be sensitive to the subwindow rectangles formed by overlapping windows. It is called when you know you really want to clear the entire screen, regardless of previous clip settings. If you want to clear the subwindow rectangles made by overlapping windows, call the just_clear function.

In clr_display, clip_work is called to set the clipping window to the size

of the entire work area, and `just_draw` is called to draw a white rectangle over the entire work area.

The simple `clip_work` function is used to get the size of the work area of the program window and then to reset the clipping window to be the entire work area.

Program 2-23. `clrdisp.c`

```
# include <gemdefs.h>
# include <obdefs.h>

clr_display(whand,vw_handle)
int whand, vw_handle;{

    clip_work( whand, vw_handle );
    just_clear( whand, vw_handle );
}
```

Program 2-24. `clipwork.c`

```
# include <gemdefs.h>
# include <obdefs.h>

# define NO_CLIP      0
# define CLIP         1

clip_work( whand, vw_handle )
int whand, vw_handle;{

    int temp[4];
    int x, y, w, h;

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    temp[0] = x;
    temp[1] = y;
    temp[2] = x + w - 1;
    temp[3] = y + h - 1;
    vs_clip( vw_handle, CLIP, temp );
}
```

The `close_all` and `close_window` Function

When the user selects the command to quit or exit a program, these functions, Program 2-25 and 2-26, take care of it.

The `close_all` function is called by `main` to close the window. `close_all` calls `close_window`, which deletes the window and draws a shrinking box for effect.

Last, `close_all` tidies up by removing the menu bar and freeing up any resources that were used for RCS data files.

Program 2-25. closeall.c

```
# include <gemdefs.h>

close_all(main_addr,whand)
struct object *main_addr;
int whand;{

    close_window( whand );
    if( main_addr )
        menu_bar( main_addr, 0 );
    rsrc_free();
}
```

Program 2-26. closewindow.c

```
# include <gemdefs.h>

close_window( whand )
int whand;{

    int x, y, w, h, foo, err;

    if( whand != -1 ){
        wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
        wind_close( whand );
        graf_shrinkbox(x+w/2,y+h/2,2,2,x,y,w,h);
        wind_delete( whand );
    }
}
```

Miscellaneous Routines and Arrays

Different startup functions are used for a desk accessory and regular application programs. In the case of a desk accessory, the Atari startup routine that initially calls GEM is called `acstart`. The startup routine for a regular program is called `gemstart`. The `acstart` routine is a much-shortened startup routine that must be supplemented with some of the variables from `gemstart` so that all the functions and variables used in the envelope library are defined.

The `accsup.c` ("accessory support") file, Program 2-27, defines several variables we need, the `brk` function, and a dummy `exit` function. The `brk` function is used to allocate memory for some of the routines in the GEM libraries. The dummy `exit` routine is included because a desk accessory might call a function that exits if it encounters an error. A desk accessory program is never supposed to exit, so the `acstart` file doesn't include `exit`. However, if `exit` is called, it must be defined, so this `exit` definition is included as a placebo for the linker, to keep it from generating an error message.

Program 2-27. accsup.c

```
# include <osbind.h>

# define BRK_SIZE      ( 256 )

long int __cpmrv      = 0;
char __pname[]        = "Accessory";
char __tname[]        = "CON:";
```


CHAPTER 2

```
char __lname[]      = "LST:";
char __xeof[]       = "\032";

char brk_mem[BRK_SIZE];
char *break = brk_mem;

brk(val)
char *val; {

    if( val < &brk_mem[BRK_SIZE-16] ) {
        _break = val;
        return(0);
    }
    return(-1);
}
_exit() { }
```

The vddata.c file, Program 2-28, defines some arrays where the VDI routines keep their input and output variables. They are only used by VDI to link; the application programs never use them directly.

Program 2-28. vddata.c

```
/*
** These are arrays that VDI should have declared for itself,
** but didn't. This is possibly because the user could declare
** them to be smaller than this if he knew he was only going to use
** functions that need the first few elements of each one.
** We have LOTS of memory on the ST, so lets not be lazy.
*/

int contrl[128];
int intin[128];
int intout[128];
int ptsin[128];
int ptsout[128];
```

Additional Envelope Functions

In addition to the functions described, the following functions and header files are part of the envelope library. addit will be discussed in Chapter 5 and newwind will be discussed in Chapter 6.

Program 2-29. addit.c

```
# include <obdefs.h>
# include <osbind.h>
# include <gemdefs.h>

# define MAXTREE      64
# define LEN          -2
# define CONSOLE      2

int next_item = 0;

addit(tree_list,parent,type,spec,x,y,w,h)
struct object *tree_list;
int parent, type, x, y, w, h;
char *spec; {

    int max_x, max_y;
```

```

extern int Wc, Hc;

if( next_item >= MAXTREE )
    return(-1);
if( w == LEN )
    w = strlen(spec);
if( tree_list[parent].ob_head == -1 )
    tree_list[parent].ob_head = next_item;
max_x = x * Wc + w * Wc;
max_y = y * Hc + h * Hc;
if( parent > 0 && max_x > tree_list[parent].ob_width ){
    printf("Parent %d's width adjusted to %d\n",parent,max_x);
    Bconin(CONSOLE);
    tree_list[parent].ob_width = max_x;
}
if( parent > 0 && max_y > tree_list[parent].ob_height ){
    printf("Parent %d's height adjusted to %d\n",parent,max_y);
    Bconin(CONSOLE);
    tree_list[parent].ob_height = max_y;
}
tree_list[next_item].ob_next = -1;
tree_list[next_item].ob_head = -1;
tree_list[next_item].ob_tail = -1;
tree_list[next_item].ob_type = type;
tree_list[next_item].ob_flags = NONE;
tree_list[next_item].ob_state = NORMAL;
tree_list[next_item].ob_spec = spec;
tree_list[next_item].ob_x = x * Wc;
tree_list[next_item].ob_y = y * Hc;
tree_list[next_item].ob_width = w * Wc;
tree_list[next_item].ob_height = h * Hc;
if( objc_add(tree_list,parent,next_item) == 0 ){
    printf("Can't add object %d to parent %d\n",next_item,parent);
}
return(next_item++);
}

```

Program 2-30. bldtree.c

```

#include <gemdefs.h>
#include <obdefs.h>

struct object *
build_tree(){
    return( 0L );
}

```

Program 2-31. clocktic.c

```

/*
** Here we handle anything we need to do when we get a clock message
** in multi(). Right now, we just ignore them. Later we might want
** to put some code here to do something automatically every so
** often.
*/

clock_ticks(wh,vw)
int wh, vw;{
}

```

Program 2-32. doclean.c

```
do_cleanup(whand,vw)
int whand, vw;
```

```
}
```

Program 2-33. gotkey.c

```
got_key(ch,whand,vw)
int ch, whand, vw;
```

```
    return(1);
}
```

Program 2-34. mousehit.c

```
# include <osbind.h>
# include <obdefs.h>
# include <gemdefs.h>
```

```
mouse_hit(butdown,x,y,kstate,num_clicks,vw)
int butdown, x, y, kstate, num_clicks, vw;
```

```
    if( butdown == 0 )
        return(1);
    return(0);
}
```

Program 2-35. newwind.c

```
# include <obdefs.h>
# include <gemdefs.h>
```

```
new_window(name,vp,hp,vs,hs,x,y,w,h,vw)
char *name;
int vp, hp, vs, hs, x, y, w, h, vw;
```

```
int whand, junk, gr;
```

```
show_mouse();
whand = setup_window(name,vp,hp,vs,hs,x,y,w,h);
wind_set(whand,WF_TOP,0,0,0,0);
clr_display(whand,vw);
return(whand);
}
```

Program 2-36. pad.c

```
pad(s1,s2,cnt)
char *s1, *s2;
int cnt;
```

```
while( cnt-- )
    if( *s2 )
        *s1++ = *s2++;
    else
        *s1++ = ' ';
    *s1 = 0;
}
```




Program 2-37. slidsize.c

```
slide_size( visible, total, size )
int visible, total, *size;{

    *size = 1000L * visible / total;
    if( *size <= 0 )
        *size = -1;
    if( *size > 1000 )
        *size = 1000;
}
```

Program 2-38. window.h

```
# define MAINMENU 0
```

Program 2-39. wfparts.h

```
# define PARTSA                                (VSLIDE:HSLIDE:UPARROW:DNARROW:LFAROW:RTARROW)
# define PARTSB                                (SIZE:MOVER:FULLER:CLOSER:NAME)
# define WF_PARTS                             (PARTSA:PARTSB)
```

How to Build the Library

All of these separate functions must be grouped into the envelope library so they can be linked with programs. The linker (link68.prg is the name of the linker in the *Atari ST Software Developer's Kit*) can look through libraries for functions and data structures that are not defined in the program. The linker then selects from the library only those files it needs. In this way, the library can contain functions that are used only if they are not defined in an application program. This is why the application-specific functions have default values that do nothing.

If the program defines one of these routines, the program's version is used instead of the default library version. If the program doesn't define an application-specific program, the linker will find a usable version in the library.

To create or add to a library using *Alcyon C*, which comes with the *Atari ST Software Developer's Kit*, we must first compile the source code files created with a text editor into object code (.O files); then we use the archiver program, called ar68.prg in the *Developer's Kit*. We build a batch file to call the archiver and to give it the names of all the compiled files we want in the library. This batch file is shown in Program 2-40. If you are using another version of C, refer to your *User's Manual* for instructions on creating a library.

The archiver program is called with the r flag (for replace), and the v flag (for verbose; omit the v flag to stop the screen chatter while the library is being created).

Then the name for our library, env.a, is given and followed by a list of filenames to be put in the library.

CHAPTER 2

We use several calls to `ar68.prg` to avoid exceeding the limit on the number of arguments to a program and also because our preferred programming style is not to wrap arguments across lines.

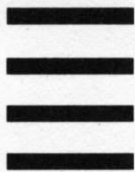
When we run this batch file by running `batch.ttp` from the desktop and giving `archive.bat` as the parameter, the file `env.a` is created. This file is ready to be used by the linker, as we shall see in the next chapter.

Program 2-40. `archive.bat`

```
ar68 rv env.a MAIN.O MULTI.O WASMSG.O ADDIT.O CLOSEALL.O
ar68 rv env.a CLRDISP.O DODISP.O DOIT.O DOMENU.O NEWWIND.O
ar68 rv env.a SETSCRN.O SETWIND.O OPENWIND.O SHOWFORM.O REDRAW.O DOARROWS.O
ar68 rv env.a CLOSIND.O CLIPWORK.O JUSTCLR.O JUSTDRAW.O
ar68 rv env.a HTOUCHED.O VTOUCHED.O SLIDEPOS.O SLIDSIZE.O OPENVWRK.O
ar68 rv env.a CLOCKTIC.O OPENDATA.O PAD.O HIDEIOUS.O GOTKEY.O MOUSEHIT.O
ar68 rv env.a POSSLIDE.O DOCLEAN.O BLDTREE.O VDIDATA.O ACCSUP.O
wait
```



3 Simple Line Graphics



3 Simple Line Graphics

■ The previous chapter described the *envelope library* which gives you a ready-made, general-purpose way to build applications programs with the GEM user interface. Using the envelope can make programming easier. The remainder of this book shows you how easy it is to write applications programs using the envelope.

The simple line graphics example demonstrating the envelope routines is a map of the world drawn on the screen. Because the envelope routines are written to be device-independent, they can draw the map in all three of the Atari's resolution modes:

low	320 × 200	16 possible colors
medium	640 × 200	4 possible colors
high	640 × 400	2 colors

We begin by customizing the routines specific to the world map program. As a brief review of Chapter 2, the nine files that "connect" the envelope library to an application follow.

config.c file defines global data items used by the library to control the look and operation of the user interface.

do_menu, defined in domenu.c, determines how the program responds when a menu item is selected.

just_draw, defined in justdraw.c, redraws the portions of the screen that are affected when a window is moved, resized, or removed from the desktop.

doit, defined in doit.c, draws the whole screen. You may sometimes decide to use it instead of the just_draw function (which is more complex) if you determine that doit can draw the screen within a second or two.

do_cleanup, defined in doclean.c, is called before a program exits to let the program close files, print score results, or reset colors to their original states. If no cleanup is required, do_cleanup does nothing.

got_key, defined in gotkey.c, is called for every keystroke received from the keyboard; it controls how the program responds to the keystroke.

build_tree, defined in bldtree.c, builds menu trees for the pull-down menus. If you use the Resource Construction Set from the *Atari ST Software Developer's Kit* to build your menus, then this routine is never called.

mouse_hit, defined in `mousehit.c`, determines what to do whenever a mouse button is pressed.

open_data, defined in `opendata.c`, opens files and does any other setup tasks necessary before the program is passed any other input.

All of these connecting routines have default versions in the library. This means you only have to modify the code of those that need to change for your particular applications program. The default versions of the unchanged routines will link in automatically.

For the world map program you only need to redefine the `config.c` file (Program 3-1) and `doit.c` (Program 3-2).

First, the file defines the window name to be "World Map." Then, because the map won't use RCS, the `USE_RCS` variable is undefined and the resource variable is set to 0.

Continuing through the program, setting the `i_am_accessory` variable to 0 indicates that this is a regular program and not an accessory. For the purpose of illustration, the accessory name variable is set to be World Map, which would appear in the Desk menu if this program were a desk accessory. Although the eight variables that control the appearance of the small (accessory) window aren't used (because this isn't an accessory program), they must still be defined for the linker to link the program properly. The program leaves them set to their default values.

The minimum window size is set to 100 pixels wide and 50 pixels high so that the smallest possible window will still be large enough for the border elements to be operable. Since a timer is not needed for this program, the timing interval is set to 0. Finally, the events variable is set to receive information about all input events.

Program 3-1. `config.c` (Map version)

```
# include <gemdefs.h>

char *wind_name          = " World Map ";

# ifdef USE_RCS
char *resource            = "WORLD.RSC";
# else
char *resource            = 0;
# endif USE_RCS

char *access_name        = " World Map ";
int i_am_accessory        = 0;
int sx                   = 20;    /* small window size */
int sy                   = 50;
int sw                   = 250;
int sh                   = 125;
int slv                  = 0;    /* small window vertical slider pos */
int slh                  = 0;    /* small window horizontal slider pos */
int svs                  = 1000; /* small window vertical slider size */
int shs                  = 1000; /* small window horizontal slider size */
int min_wide             = 100;
int min_high             = 50;
int interval             = 0;
int events               = MU_MESAG | MU_BUTTON | MU_KEYBD | MU_M1 | MU_M2;
```


The doit Function for the Map Application

The code below for the doit function is so simple that you may wonder why it is needed for this program.

This version does only one thing: calls the map function, which is where all the map-drawing work is done. The reason doit is included in the map program is in case you want to expand it to do something more with the map after it is drawn.

Program 3-2. doit.c

```
doit (whand, vw)
int whand, vw; {

    map (whand, vw);
}
```

Drawing the Map

The map is drawn by connecting a series of points outlining the coastlines of the world's continents and main islands. The data defining the points is in a file called world.c. That data is a long series of numbers that has been packed in order to save space (and typing). The packing method is: multiply each *y* coordinate by 640 (the number of columns on the screen) and then add the *x* coordinate value to it. This converts each point's *x* and *y* coordinates into one integer, which is why the data for the points does not appear to have *x* and *y* coordinates. To make the data a bit more manageable, continuous lines are marked with -1 at their beginning.

The data for world.c is listed in the Appendix. If you decide to type the data in instead of purchasing the disk with the data, you may want to draw only certain coastlines to save time (and finger fatigue, not to mention boredom). You can type in any part of the data you wish; just be sure you type in an entire segment. Each segment begins with a -1 and continues up to, but not including, the next -1. Type in as many segments as you wish. As you look through the data segments you'll notice that they have been printed in order of size, starting with the largest coastlines. Remember, though, that the more segments you type in, the more of the world will appear on the screen.

Program 3-3 draws the map.

In the program, map calls the GEM routine v_pline, which plots lines between sets of points in an array. Since there are a great many points, we want to find out how many points v_pline will accept before we call it.

The GEM vq_extnd function tells us the number of points, as well as a lot of other information we don't need to know right now. It returns 57 values into the array we pass to it. What those values are depends on the second argument in the parameter list. We have defined WORKVALS and INQUIRE to values that point to the values we want. If vq_extnd is given the argument WORKVALS, the function gives us the same output as opnvwk in multi. If given the INQUIRE value, the function returns 19 new values (and zeros filling

out the array to 57). With these statements the fourteenth item in the INQUIRE array is located, and that item indicates how many vertices `v_pline` can accept.

```
vq_extnd( vw, INQUIRE, info );  
max = info[14];
```

This corresponds to a little more than 100 points on the computer. Larger systems may differ, as may future machines or ROMs. Note that because of the data packing, points are constructed from two values: one each for the *x* and *y* coordinates.

The program starts unpacking the data in the world array, converting the packed numbers back into the original *x* and *y* coordinates. A concise method of unpacking the coastline data should be possible; however, due to some bugs in the way the compiler handles long integers, some creative programming is necessary. To work around the bugs, you can define two temporary variables, *row* and *col*, and do each arithmetic operation on a separate line, avoiding the `%` operator on long integers. This is not a recommended coding style, but serves a short-term purpose until the bugs are fixed.

As each integer in the world array is converted to its two corresponding values, each of the values is placed into the points array and the count variable is incremented by one.

Then the `v_pline` routine is called and given the handle of the virtual workstation, the number of points to be passed to it (one point is two values; therefore we divide the count by 2), and the address of the array where the *x* and *y* values are stored.

If `v_pline` could accept a large number of vertices, you could just pass it all the points between the `-1` markers, calling `v_pline` once for every `-1` found in the array. But since `v_pline` accepts just over 100 points, a second *if* statement is added to handle long coastlines. The count variable, whose value equals the number of values in the points array, is compared to the maximum points `v_pline` will accept—again multiplied by two since there are two values per point.

Program 3-3. map.c

```
# include <gemdefs.h>  
# define FUDGE_X      640L  
# define FUDGE_Y      400L  
# define WORKVALS      0  
# define INQUIRE      1  
static int points[512];  
  
map(whand,vw)  
{  
    int whand, vw;  
  
    extern long int world[];  
    unsigned int info[60];  
    int wx, wy, ww, wh;  
    static int old_w = -1, old_h = -1;  
    long int row, col;  
    register int x, count, max;
```

```

wind_get( whand, WF_WORKXYWH, &wx, &wy, &ww, &wh );
if( ww == old_w && wh == old_h )
    return;
old_w = ww;
old_h = wh;

clr_display(whand,vw);
vq_extnd( vw, INQUIRE, info );
max = info[14];                                /* max points per v_pline */
hide_mouse();
count = 0;
for( x = 0; world[x]; x++ ){
    if( world[x] == -1 ){
        if( count )
            v_pline( vw, count/2, points );
        count = 0;
        continue;
    }
    if( count >= (max-1)*2 ){
        if( count )
            v_pline( vw, count/2, points );
        x--;                                /* use last vertex again */
        count = 0;
    }
    /* Compiler bugs prevent this from working as originally written.
    ** (Alcyon C compiler, very early developer's kit)
    ** This version of the compiler cannot do remainder operations on
    ** long integers, and cannot do more than one or two long integer
    ** operations in one expression without getting confused.
    **
    points[count++] = wx + ((world[x] % FUDGE) * ww) / FUDGE;
    points[count++] = wy + ((world[x] / FUDGE) * wh) / FUDGE;
    */
    row = world[x] / FUDGE;
    col = world[x] - row * FUDGE;
    col %= ww;
    row %= wh;
    col /= FUDGE;
    row /= FUDGE;
    points[count++] = wx + col;
    points[count++] = wy + row;
}
show_mouse();
}

```

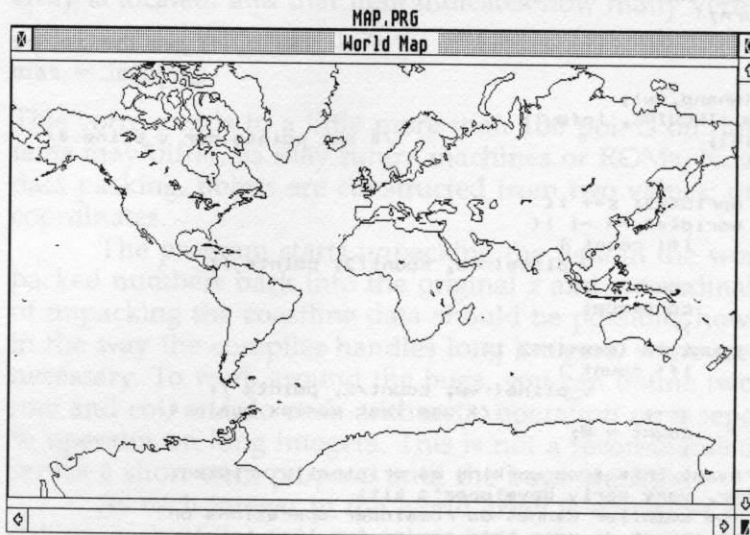
Buy an ST and See the World

You can now look at what happens in the program after it has been compiled, and see what the envelope library does for simple programs. Figure 3-1 shows what the program will display.

The first thing to appear when you run the program is the border area. A title bar appears at the top, with a Close box in the upper left corner, a Full box in the upper right corner, and a Size box in the lower right corner. Sliders with arrows at either end appear; there are no gray areas visible because the slider box is filling the whole area. If you click on the slider boxes or arrows, the sliders flash, but nothing else happens.

The program then draws the map, starting with the long coastlines and then outlining the lakes and islands.

To see GEM in action, place the mouse on the Size box in the lower

Figure 3-1. A Full-Screen Map

right corner and hold the mouse button down; collapse the window by moving it to the upper left corner. GEM adds the “rubber band” window frame for some nice visual feedback. When you release the button, the map window shrinks to its smallest size and the map is redrawn, scaled down to the tiny window size.

Place the mouse on the title *World Map*, and drag the window to the center of the screen. Again, the map will redraw.

Try dragging the Size box to make a long skinny window. The map will stretch and constrict to fit in the window, as if it were drawn on a stretched sheet of rubber (see Figure 3-2).

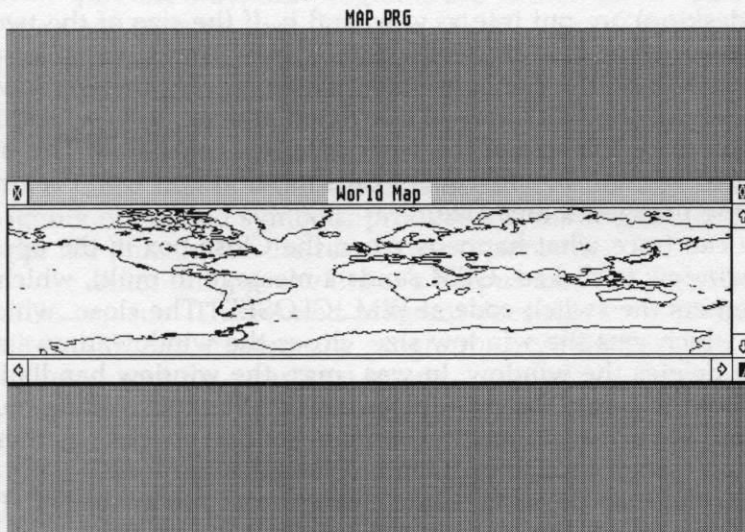
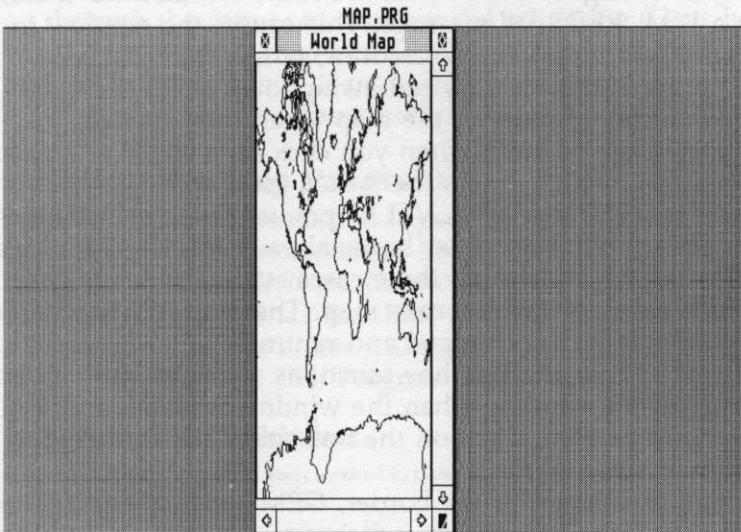
To understand the part the envelope library plays in all of this activity, you need to examine which library functions are operating and the parts they are playing.

The main routine has called `setup_window`, which calls `open_window` to put up the window. The `open_window` routine creates the window using the constant `WF_PARTS` to define the border areas, and then adds the name and sliders. The slider boxes fill the area because main passed the maximum value 1000 through `setup_window` to `vsize` and `hsize`.

The main function calls the multi function. After the Size box is dragged to the upper left corner and released, multi receives a message from GEM that something has happened in the border areas. multi calls `was_msg` with the contents of the message.

The switch code in `was_msg` is entered at the `WM_SIZED` entry point. The minimum window size is checked; then the current size is set to the values

Figure 3-2. The proportions of the map adjust to fit the dimensions of the window.



passed from GEM. Using these values, the `was_msg` subroutine calculates the size of the work area and sets it. It then redraws the display.

If the window is enlarged, GEM sends the same `WM_SIZED` message, but follows it with a `WM_REDRAW` message. This causes the window to be redrawn again because `was_msg` calls `do_redraw`, which calls `just_draw`, which calls `doit`. To prevent unnecessary redrawing, `map` saves the width and height of the window and only redraws the map when they change.

GEM sends a message to `multi` when you click on a slider or arrow. `multi` sends it to `was_msg`, which enters the switch code at the `WM_ARROWED`, `WM_HSLID`, or `WM_VSLID` points. These call routines to move the sliders, but since our map slider boxes already fill the slider area, nothing happens. The switch code in all three cases starts a series of calls: They call `do_display`, which calls `doit`, which calls `map`. The `map` function determines that the window size hasn't changed and returns.

Take a look now at how the Full box functions. Click on the Full box in the upper right corner of the window when the window is small, and it quickly fills the entire desktop work area. Click on the box again and the window jumps back to its former size.

Because the Full box is part of the border, GEM sends a message to `multi`, which passes it to `was_msg`, which switches to the `WM_FULLED` entry point. Here the size and position of the current application window are put into the `xc`, `yc`, `wc`, and `hc` variables. The size and position of the desktop work area (window 0 is the desktop) are put into `x`, `y`, `w`, and `h`. If the size of the two windows match, the window is in its "full" state. `x`, `y`, `w`, and `h` variables are set for the current size to be the previous state, which is what the window is to become. Then the size of the work area is calculated and set. `x`, `y`, `w`, and `h` variables are also set to be the size of the work area just calculated. This step seems redundant, but without it, subsequent calls to `wind_get` won't return the correct values for the new work area size.

Finally, you can trace what happens when the Close box in the upper left corner of the window is clicked. GEM sends a message to `multi`, which calls `was_msg`, which enters the switch code at `WM_CLOSED`. The `close_window` function is called, which gets the window size, closes the window, draws a shrinking box, and deletes the window. In `was_msg`, the window handle is set to `NO_WINDOW` and the `exit_flag` variable is set to `BYE_BYE`, telling `multi` to break out of its loop and return to main. The main routine checks whether this program is a desk accessory and, since it isn't, terminates the loop.

At the end of the map drawing, some cleanup routines are called. The `do_cleanup` function is called, but since it is the default library version, it does nothing. Then `close_all` is called, which calls `close_window`, which also does nothing because `was_msg` has set the window handle to `-1`. Then the GEM routines `v_clswnk` and `appl_exit` are called to close the virtual workstation and exit the program.

Users can also exit the program in two other ways. Pressing any key on the keyboard will cause the program to exit, since our library default version of `got_key` always returns 1, which means exit. Pressing a key will cause `multi` to go through the same procedure it did when the Close box was activated. Likewise, if there are menus and QUIT is clicked, `do_menu` will return 1 for exit, and the same path out of `multi` will be followed.

Later programs show both of these exiting methods in more detail.

Building the Map Program

After the four subroutines that generate the map are compiled, they are ready to be linked with the envelope library to build the map program. To link them, using the *Atari ST Software Developer's Kit*, we build two files: `linkit.bat`, which contains the commands that `batch.ttp` will execute, and `link.arg`, which contains a list of the files the linker will link.

The `linkit.bat` file, Program 3-4, is only four lines long.

Program 3-4. `linkit.bat`

```
c:\bin\link68 [undefined,symbols,command[link.arg]]
c:\bin\relmod a
c:\bin\rm a.68k
c:\bin\wait
```

By the way, our programs are written for an ST with a hard disk and we keep our tools on disk C: in the folder `\bin`. If your system configuration differs (no hard disk, for instance), you'll need to change the lines in `linkit.bat` accordingly—for example,

A:link68[args].

The first line calls the linker program, `link68`, and includes arguments to ignore undefined symbols, produce a symbol table for the debugger, and read the file `link.arg` for the rest of the commands.

After linking, the linker's output must be converted into a program that TOS can run. The `relmod` program does this by reading the linker output in the file `a.68K` and creating the program `a.prg`. We then remove the temporary `a.68k` file and wait for a carriage return.

The second file, `link.arg` (Program 3-5), is only three lines long.

Program 3-5. `link.arg`

```
a.68k=c:gemstart.o,main.o,
CONFIG.O,DOIT.O,MAP.O,WORLD.O,
env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```

The first line tells the linker to create the file `a.68k` and to put the file `gemstart.o` at the beginning of the program, followed by `main.o`.

The second line lists the four files created in this chapter for the map

CHAPTER 3

program. Since the linker is not sensitive to upper- or lowercase, the filenames are distinguished by uppercase.

The last line is the list of libraries necessary to link with our map sub-routines, starting with the envelope library created in Chapter 2.

After `linkit.bat` and `link.arg` have been constructed, the program is linked by clicking on `batch.ttp` and typing `linkit` as the argument. Messages appear on the screen as `A.PRG`, the default name for the executable program file, is created. When the map program has been linked, you may want to rename `A.PRG` to `WORLDMAP.PRG` from the desktop menu. Then open `WORLDMAP.PRG` by clicking on it, and see the world.



4 Business Graphics

4 Business Graphics

■ This chapter explores the Virtual Device Interface (VDI) part of GEM. The VDI routines are responsible for activities that involve input/output devices, such as converting coordinates for a printer or the screen, writing to a disk, or certain basic graphics operations like circles, lines, and fill.

The program developed here prints presentation-quality line graphs, bar charts, and pie charts. It will draw thin and wide lines with varying endpoints; it will also draw rectangles and pie-chart slices, and scale the output to fit any window size and resolution. In addition, you'll see how to fill pie slices and the bars in a bar chart with various patterns such as stripes, bricks, hatches, and so forth.

In the course of showing how to create business graphs and charts, we'll also demonstrate how to open a file and read data from it. Two ways of giving a filename to the program are shown: The first is when the filename is entered as the parameter in a .TTP dialog when the user starts the program, and the second is from a file-selector dialog box that opens after the program is started. The two types of dialog boxes are shown in Figure 4-1.

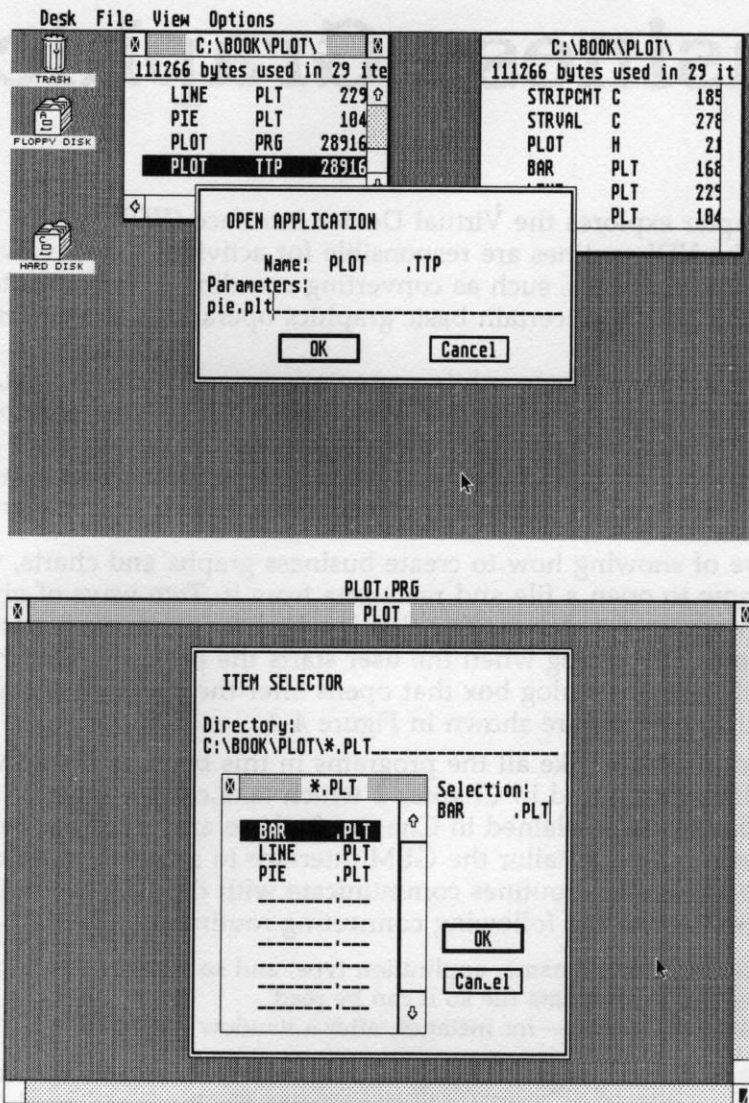
This graphics program, like all the programs in this book, is linked with the library of routines introduced in Chapter 2 which take care of most of the GEM-interface behavior. As explained in Chapter 2, there are usually some connecting routines required to tailor the GEM interface to a specific application, and to let the application routines communicate with the GEM routines. For this program we'll write the following connecting routines:

config.c	to give the window name, application type, and so on
open_data.c	to find and open a data file so it can be read
doit.c	to redraw the screen—for instance, after a window is resized

When all parts of the program are finally linked, there will be the connecting routines, the application routines that generate the business graphics from the data given, and the envelope library routines (in the file named env.a).

The envelope routines listed in Chapter 2 must be compiled and linked into the env.a library before they can be used with this application program.

Figure 4-1. The Two Types of Dialog Boxes



The config.c File

As is normal, the config.c file requires a few changes for this application. Program 4-1 indicates the changes.

The window name is set to read "Plots and Charts" and the `i_am_` accessory variable is set to 0, since this will be a regular program (not a desk-accessory program). Neither the Resource Construction Set (RCS) nor the acces-

sory window variables are used, but they are filled just in case we decide to change things later. Filling these variables is optional, but the variables must be present or the program won't link.

Program 4-1. config.c

```
# include <gemdefs.h>

char *wind_name          = " Plots and Charts ";

# ifdef USE_RCS
char *resource            = "PLOTS.RSC";
# else
char *resource            = 0;
# endif USE_RCS

char *access_name        = " Plotting ";
int i_am_accessory       = 0;
int sx                   = 20; /* small window size */
int sy                   = 50;
int sw                   = 250;
int sh                   = 125;
int slv                  = 0; /* small window vertical slider pos */
int slh                  = 0; /* small window horizontal slider pos */
int svs                  = 1000; /* small window vertical slider size */
int shs                  = 1000; /* small window horizontal slider size */
int min_wide             = 100;
int min_high             = 50;
int interval             = 0;
int events = MU_MESAG : MU_BUTTON : MU_KEYBD : MU_M1 : MU_M2;
```

Opening and Reading a Data File

The first step is to read the file containing the data to be plotted. This is done with the functions `open_data`, `select_file`, and `read_data`.

The `open_data` function. The `open_data` function, Program 4-2, is one of the programs that connect the application and the envelope library, and is called by the main routine in the envelope library. If a filename exists, it is passed to `open_data` in the file argument.

The filename can be passed to main as a parameter when the program is started for either a .TTP program, or from COMMAND.TOS. If the argument passed in the file variable is a filename, then the file is immediately opened and read.

However, if there is no argument, then the file variable is passed as "", the null string, and the user must be prompted to enter a filename. This is done with a file-selector dialog box, such as the one shown in Figure 4-1.

`open_data` calls the `select_file` function to produce this part of the user interface. (`select_file` is discussed below.) `select_file` displays a dialog box and a list of filenames, if any are present, and allows the user to select or enter a filename. This filename is put into the file variable, and then the GEM VDI function `fopen` tries to open it. If `fopen` can't locate the file—for example, the user entered the wrong filename or the file is on a different disk from the one in the drive—`select_file` is called again to prompt the user for another filename. This

continues until a file can be opened or the user selects the CANCEL button. Selecting CANCEL causes `select_file` to return 0, which causes the program to exit in the main function.

Once the file is open, `open_data` calls `read_data` to read it.

Program 4-2. `opendata.c`

```
# include <stdio.h>

int data_count;

open_data(whand,vw,file)
int whand, vw;
char *file;{

    extern int errno;
    FILE *fd, *fopen();

    if( file[0] == 0 ){
        if( select_file(file) == 0 )
            return( 0 );
    }

    while( (fd = fopen(file,"r")) <= 0 ){
        form_error(errno);
        if( select_file(file) == 0 )
            return( 0 );
    }

    return( data_count = read_data( whand, vw, fd ) );
}
```

The `select_file` function: Getting the filename. The purpose of this function, Program 4-3, is to display the file-selector dialog box and return the name of the file the user enters to `open_data`, so the data in the file can be read.

These few lines of code accomplish quite a bit. They use GEM's `fsel_input` function, which puts up a dialog box, fills it with filenames to select from, and accepts the user's input. The directory path and an array into which to put the filename must be passed to `fsel_input`.

The directory path is really entered as a search pattern. To build the search pattern we get the current disk drive with the GEM `Dgetdrv` function, which returns 0 for drive A, 1 for drive B, and so on. The function gets the current working directory pathname with the GEM `Dgetpath` function—for example, `\GAMES\STARTREK` or `\PROJECTS\BOOK\CHAP1`. `Dgetpath` is passed an array, `curdir`, in which to put the path. The drive returned by `Dgetdrv` is also passed, with 1 added to it because `Dgetpath` names the drives starting at 1 for A, and so on.

Now that the program has handled all the pieces for the path, the GEM `sprintf` function is called to put them together into a string such as

A:\PROJECTS\PLOT*.PLT

The drive number is given as an argument to `sprintf`, converted to its corresponding letter in the `drv` variable, and the pathname is stored in the `curdir` variable.

sprintf returns a string which is passed to the GEM function `fsel_input`. This function then lists, in the dialog box, all the files in the directory `\PROJECTS \PLOT` on drive A that have the `.PLT` extension. The user can then select from the list by clicking on the filename with the mouse. The selected filename is placed in the `newfile` array, and whichever button the user has picked, OK or CANCEL, is returned in the button variable. If for some reason `fsel_input` should encounter an error, 0 is returned to indicate failure. Also, 0 is returned to indicate failure to select a file if the user happens to pick the CANCEL button.

If the user has selected a filename or typed one in and clicked the OK button, its name is copied from the `newfile` array to the `file` array with the GEM `strcpy` routine. The `file` variable is returned to `open_data` and 1 is returned to indicate success.

Program 4-3. `slctfile.c`

```
#include <osbind.h>

# define CANCEL      0
# define OK          1

select_file( file )
char *file;{

    int button, drv;
    char dir[80], newfile[80], curdir[80];

    drv = Dgetdrv();
    Dgetpath(curdir,drv+1);
    sprintf( dir, "%c:%s\\$.PLT", drv+'A', curdir );
    newfile[0] = 0;
    if( fsel_input( dir, newfile, &button ) == 0 )
        return( 0 );
    if( button == CANCEL )
        return( 0 );
    strcpy( file, newfile );
    return( 1 );
}
```

The `read_data` Function: Reading the Data File

After `open_data` has opened the data file returned by `select_file` (unless the user entered the filename as a parameter when the PLOT program was started), it calls the `read_data` function to read it.

A data file for use with the PLOT program is structured like this:

Figure 4-2. A Sample Data File

Corporate Profits
LINE

1.0,	1.0	# x & y grid increments: don't draw
0.0,	0.0	# lower left point: don't draw
10.0,	10.0	# upper right point: don't draw
1.0,	1.0	
3.0,	7.4	
4.2,	4.0	
5.0,	9.3	
6.5,	6.0	
7.0,	8.5	
8.7,	0.5	

CHAPTER 4

The first line always contains a title and the second line always states the chart type. Subsequent lines contain data whose structure is specific to a chart type. Labels are enclosed in double quotes, and any characters following a # on a line are comments which `read_data` ignores.

The `read_data` function, Program 4-4, will read this data and pass it to the right charting routine.

`read_data` first looks for the chart title and type, displaying error messages with `show_form`, and returning 0 if they cannot be read. The title line is passed to the GEM routine `wind_set`, which takes care of displaying it in the window's title bar.

The function expects the chart type to be LINE, BAR, or PIE in the second line of the data file. If it isn't one of these three, a message is displayed, and the type defaults to a LINE chart.

Next, the GEM `fgets` function gets lines of data from the file. For each line, the program calls the `extract` function to get the individual data items out of the line. `extract` handles multiple items on a line by returning after each one, leaving a pointer to the next character to be read. `read_data` determines if the next character is a null byte (end of line) and calls `extract` again if it isn't. With this method, it is not necessary to tell the `read_data` and `extract` functions how many numbers are on a line, since they are handled one at a time until the end of the line is reached. As each data item is read, it is put into the `data_set` array in the place pointed to by the `dp` (data pointer) variable. Label data items are returned in the `labels` and `labcnt`, the index into the `labels` array, is incremented.

Program 4-4. `readdata.c`

```
# include <stdio.h>
# include <gemdefs.h>

# define MAX_DATA      100
# define LINE_CHART    1
# define PIE_CHART     2
# define BAR_CHART     3

char title[80];
double data_set[MAX_DATA];
int plot_type, labcnt;
char *labels[MAX_DATA];

read_data( whand, vw, fd )
int whand, vw;
FILE *fd;{

    char line[128], type[128], *lp, *extract(), *fgets();
    int x;
    double *dp;

    if( fgets( title, sizeof(title), fd ) != title ){
        show_form( "Missing title line!");
        return( 0 );
    }
```



```

strip_comment( title );
wind_set( whand, WF_NAME, title, 0, 0 );
if( fgets( type, sizeof(type), fd ) != type ){
    show_form( "Missing type line!" );
    return( 0 );
}
strip_comment( type );
if( strcmp( type, "LINE" ) == 0 )
    plot_type = LINE_CHART;
else if( strcmp( type, "PIE" ) == 0 )
    plot_type = PIE_CHART;
else if( strcmp( type, "BAR" ) == 0 )
    plot_type = BAR_CHART;
else {
    sprintf( line, "Unknown type '%s': assuming 'LINE'", type );
    show_form( line );
    plot_type = LINE_CHART;
}
dp = data_set;
labcnt = 0;
for( x = 0; x < MAX_DATA; ){
    if( fgets( line, sizeof(line), fd ) != line )
        break;
    strip_comment( line );
    lp = line;
    while( *lp ){
        lp = extract( lp, dp, &labels[labcnt] );
        if( labels[labcnt] )
            labcnt++;
        x++;
        dp++;
    }
}
return( x );
}

```

The strip_comment function. In the course of each data line being read, comments and trailing white spaces are removed by the strip_comment function, Program 4-5.

This function scans a string until it finds a # or the end of the string, and then backs up, replacing any blanks or tabs with null characters.

Program 4-5. stripcmt.c

```

strip_comment( str )
char *str;{
    while( *str && *str != '#' && *str != '\n' )
        str++;
    while( *str == '#' || *str == ' ' || *str == '\t' || *str == '\n' )
        *str-- = 0;
}

```

The extract function: Placing the data into an array. The extract function, Program 4-6, breaks down each line of data from the file into the individual data items that are separated by spaces, and puts the items into an array for numbers and an array for labels.

The first nonwhite character is found with a two-line loop that skips blanks and tabs. Having found a character, the program checks to see if it is a

double quote, and if it is, all the characters up to the next double quote are collected and a pointer is put in the *label variable that points to the place in memory allocated to the block of characters. The standard C functions, strlen and malloc, are used to set up the memory block and return the pointer. The second double quote is set to 0 to mark the end of the string. If the character is not a double quote, *label remains 0 (which was set at the start of this function). Finally, the C function strcpy is used to copy the string into the label array in the allocated position in memory.

To handle the data items that are not labels, a loop is used to collect all characters up to a comma, null character, or newline character, and then the program calls the ASCII-to-floating point function atof to convert the character string into its corresponding binary form. The binary number is put into the proper element of the data_set array by the pointer, dp.

Finally, the pointer to the the next unread character in the line is returned to read_data to determine whether extract should be called again, or if it has finished reading the file.

Program 4-6. extract.c

```
# define DQUOTE      '\\"'

char *
extract( str, dp, label )
char *str;
double *dp;
char **label;{

    register char *p;
    char *malloc();
    double atof();

    while( *str == ' ' || *str == '\t' )
        str++;
    *label = 0;
    if( *str == DQUOTE ){
        p = ++str;
        while( *str != DQUOTE && *str && *str != '\n' )
            str++;
        if( *str )
            *str++ = 0;
        else
            *str = 0;
        *label = malloc( strlen( p ) + 1 );
        strcpy( *label, p );
        if( *str == ',' )
            str++;
        while( *str == ' ' || *str == '\t' )
            str++;
    }
    for( p = str; *p && *p != ',' && *p != '\n'; p++ )
        ;
    if( *p == ',' )
        *p++ = 0;
    else
        *p = 0;
    *dp = atof( str );
    return( p );
}
```

The doit Function: Drawing a Specific Type of Chart

Recall that the second item in the data file is the chart type. In the `open_data` function of the chart program, the `plot_type` variable is set to `LINE`, `BAR`, or `PIE`. In `open_data` the global variables `data_set` and `data_count` are also set. Here these variables are passed to another one of the routines that connect our application program to the envelope library and GEM—the `doit` function, Program 4-7.

Several routines in the envelope call the `doit` function whenever they need to redraw the screen. `doit` clears the screen and calculates the size of a rectangle that nearly fills the window's work area. The rectangle consists of four global variables—`box_x`, `box_y`, `box_w`, and `box_h`—that define `x`, `y`, `w`, and `h`, and are used by routines called by `doit`. Chapter 2 discusses another similar `doit` function in detail. For now, notice that `doit` switches on `plot_type`, and then the data that `open_data` put into the `data_count` and `data_set` arrays is passed to one of three subroutines: `line_chart`, `bar_chart`, or `pie_chart`. These routines actually do the work of drawing the screen.

Program 4-7. `doit.c`

```
# include <gemdefs.h>

# define LINE_CHART    1
# define PIE_CHART     2
# define BAR_CHART     3

int box_x, box_y, box_w, box_h;

doit( whand, vw )
int whand, vw;{

    extern int gl_wchar, gl_hchar, plot_type, data_count;
    extern double data_set[];
    int xwork, ywork, wwork, hwork;

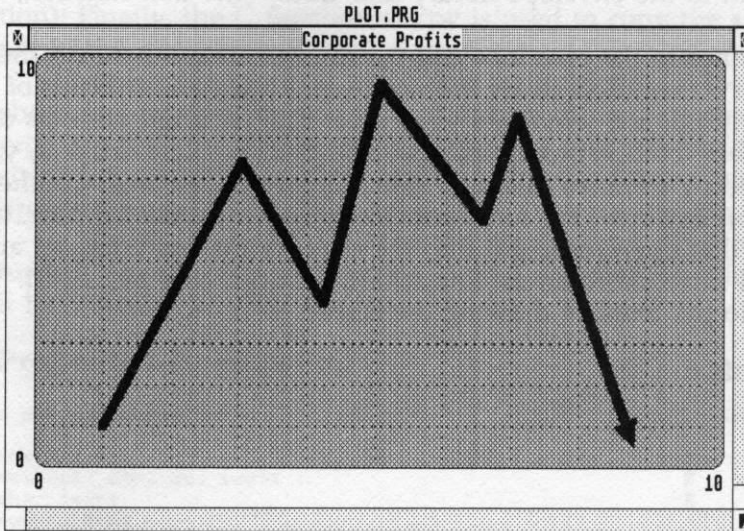
    clr_disp( whand, vw );
    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    hide_mouse();
    box_x = xwork + gl_wchar;
    box_y = ywork + 2;
    box_w = wwork - gl_wchar * 2;
    box_h = hwork - gl_hchar * 2;

    switch( plot_type ){
        case LINE_CHART:
            line_chart( vw, data_count, data_set );
            break;
        case BAR_CHART:
            bar_chart( vw, data_count, data_set );
            break;
        case PIE_CHART:
            pie_chart( vw, data_count, data_set );
            break;
    }
    show_mouse();
}
```

The line_chart Function: Drawing a Line Chart

The line_chart function will create the line-chart style of graph as shown in Figure 4-3.

Figure 4-3. A Sample Line Chart



The data to produce the chart shown in Figure 4-3 is in the file line.plt, Program 4-8.

After the chart type, the structure of the data in a chart data file should contain at least three pairs of numbers to be used for setting up the plot. Following these are the data pairs, which define the vertices of the graph.

The chart type, LINE, causes doit to run the line_chart function, Program 4-9.

Since line_chart takes its data in paired x and y coordinates, it first divides the data count (passed to it by do it) by 2. It then verifies that there are at least three pairs of numbers.

The first pair are the x and y distances between the lines of the grid that we'll put behind the graph line. These units are the same as for the rest of the data, so you can think of the grid lines as occurring every one million dollars if the graph data is in millions of dollars.

The next two pairs of numbers define the minimum and maximum values for the graph. This feature lets us improve the graph aesthetically by making the edges of the graph area extend past the graph lines, thereby keeping the graph lines from touching the edges of the graph area.

If the value for the x increment that was given in the first pair of numbers equals the maximum x value in the second pair of numbers, then only horizontal lines will be drawn behind the graph.

The remaining data pairs are interpreted as x and y coordinates that define the dots to be connected with the `v_pline` routine, as is done with the `map` program in Chapter 3.

As the `line_chart` function continues to execute, it calls several other functions we provide, and some functions provided by VDI. The next few paragraphs summarize these calls and their purposes.

First, the `range` function is called to verify that the correct minimum and maximum values are set, to prevent the graph lines from extending beyond the edge of the chart.

The `grid` function is next, putting up the background grid for the graph. This function is used to convert the graph points from the graph units, such as millions of dollars, to pixel units. Once the set of graph points is in pixel units, some VDI functions are called to plot them.

The line mode is set to `SOLID` by calling the VDI `vs_ltype` function. Several "poly-line" line types are defined at the beginning of the `line_chart` file, so you can experiment with the line mode setting if you wish.

You can also experiment with the line width, vertices markers (poly-markers), and end shapes. VDI's `vsm_type` sets the type of marker for the GEM `v_pmarker` routine to use. `v_pmarker` is just like `v_pline`, but it puts distinct markers at the vertices instead of connecting the points with lines. We use `vs_lwidth` to set the width of the lines. Thick lines set to eight pixels wide show up well on overhead slides.

Last, the GEM VDI function `v_pline` is called, which plots the points and connects them with lines in the style we indicated in the previous statements.

Program 4-8. `line.plt`

```
Corporate Profits
LINE
1.0,    1.0          # x & y grid increments: don't draw
0.0,    0.0          # lower left point: don't draw
10.0,   10.0         # upper right point: don't draw
1.0,    1.0
3.0,    7.4
4.2,    4.0
5.0,    9.3
6.5,    6.0
7.0,    8.5
8.7,    0.5
```

Program 4-9. `linechrt.c`

```
/*
** Poly-marker shapes
*/
# define DOT      1
# define PLUS     2
# define STAR     3
```

CHAPTER 4

```
# define SQUARE          4
# define CROSS           5
# define DIAMOND          6

/*
** Poly-line line types
*/
# define SOLID            1
# define LONGDASH         2
# define DOTTED           3
# define DASHDOT          4
# define DASH             5
# define DASHDOTDOT       6

/*
** Poly-line end styles
*/
# define SQUARE          0
# define ARROW            1
# define ROUNDED          2

line_chart( vw, count, data )
int vw, count;
double *data;{

    double max_x, max_y, min_x, min_y, x, y;
    int i, points[512], off;
    extern int box_x, box_y, box_w, box_h;

    count /= 2;
    if( count < 3 ){
        show_form("No data after increments and corners");
        return;
    }
    x = *data++;
    y = *data++;
    count--;
    range( count, data, &max_x, &max_y, &min_x, &min_y );
    off = grid( vw, x, y, max_x, max_y, min_x, min_y );
    data += 4; /* skip corner data */
    count -= 2;
    for( i = 0; i < count * 2; i += 2 ){
        points[i] = box_x +
            scale(data[i],max_x,min_x,box_w-off) + off;
        points[i+1] = box_y + box_h -
            scale(data[i+1],max_y,min_y,box_h);
    }
    vsl_type( vw, SOLID ); /* solid lines */
# ifdef THIN_LINES
    vsm_type( vw, DIAMOND ); /* diamonds at vertices */
    v_pmarker( vw, count, points ); /* draw the diamonds */
    vsl_width( vw, 1 ); /* draw thin lines */
# else
    vsl_width( vw, 8 ); /* draw wide lines */
    vsl_ends( vw, ROUNDED, ARROW ); /* with fancy ends */
# endif THIN_LINES
    v_pline( vw, count, points ); /* draw the lines */
}
```

The range Function

The range function, Program 4-10, is called by line_chart to determine the largest and smallest coordinates. The function expects them to be the minimum and maximum values that were the first two number pairs in the data file.

However, in case an error has been made and one of the other data values is larger or smaller, this function resets the maximum and minimum x and y variables to the correct values. This prevents the graph line from extending beyond the window.

The range function loops through the data array, checking each pair of numbers against the current minimum and maximum x and y values. If a number is found that is less than the minimum, the minimum is set to the number. The same process is done for maximum numbers. When the routine is finished, `max_x` and `max_y` contain the largest x and y values and `min_x` and `min_y` contain the lowest x and y values.

Program 4-10. range.c

```
range( count, data, max_x, max_y, min_x, min_y )
int count;
double *data, *max_x, *max_y, *min_x, *min_y;{

    int i;

    *min_x = *max_x = data[0];
    *min_y = *max_y = data[1];
    for( i = 0; i < count * 2; i += 2 ){
        if( data[i+0] < *min_x )
            *min_x = data[i+0];
        if( data[i+0] > *max_x )
            *max_x = data[i+0];
        if( data[i+1] < *min_y )
            *min_y = data[i+1];
        if( data[i+1] > *max_y )
            *max_y = data[i+1];
    }
}
```

The grid Function and Its Calls

The `line_chart` function calls `grid`, Program 4-11, to label the axes, draw a colored rectangular background with rounded corners and a pattern so it is suitable for either a color or monochrome monitor, and draw dotted lines at intervals to mark the graph divisions.

`grid` calls the `label` function, Program 4-12, to label the upper and lower left corners of the graph with the maximum and minimum values.

The `label` function converts the stored floating-point values of the minimum and maximum numbers back into text strings with the `strval` function, Program 4-13, which follows it.

Program 4-11. grid.c

```
# include <obdefs.h>

/*
** Poly-marker shapes
*/
# define DOT          1
# define PLUS         2
# define STAR         3
# define SQUARE      4
```

CHAPTER 4

```
# define CROSS          5
# define DIAMOND        6

/*
** Poly-line line types
*/
# define SOLID          1
# define LONGDASH        2
# define DOTTED          3
# define DASHDOT         4
# define DASH            5
# define DASHDOTDOT      6

/*
** Poly-line end styles
*/
# define SQUARE         0
# define ARROW           1
# define ROUNDED         2

grid( vw, x, y, max_x, max_y, min_x, min_y )
int vw;
double x, y, max_x, max_y, min_x, min_y;{

    int i, mode, line[16], off, off2, x_inc, y_inc;
    char str[80];
    extern int box_x, box_y, box_w, box_h, gl_wchar, gl_hchar;

    off = label( vw, min_y, box_x, box_y+box_h );
    off2 = label( vw, max_y, box_x, box_y+gl_hchar+1 );
    if( off2 > off )
        off = off2;
    off *= gl_wchar;          /* convert chars to pixels */

    if( x < max_x ){
        label( vw, min_x, box_x+off, box_y+box_h+gl_hchar );
        off2 = strval( max_x, str ) * gl_wchar;
        label( vw, max_x, box_x+box_w-off2, box_y+box_h+gl_hchar );
    }

    draw_box( vw, box_x+off, box_y, box_w-off, box_h );
    vsl_width( vw, 1);
    vsl_color( vw, 2);
    vsl_type( vw, DOTTED );          /* grid made of dotted lines */
    vsl_ends( vw, SQUARE, SQUARE ); /* with simple ends */
    mode = get_mode(vw);             /* save old mode */
    vswr_mode( vw, MD_TRANS );       /* transparent between dots */
    if( x < max_x ){
        x_inc = scale( x, max_x, min_x, box_w-off );
        for( i = x_inc; i < box_w-off; i += x_inc ){
            line[0] = i + box_x + off;
            line[1] = box_y;
            line[2] = i + box_x + off;
            line[3] = box_y + box_h;
            v_pline( vw, 2, line );
        }

        y_inc = scale( y, max_y, min_y, box_h );
        for( i = y_inc; i < box_h; i += y_inc ){
            line[0] = box_x + off;
            line[1] = box_y + box_h - i;
            line[2] = box_x + box_w - off;
            line[3] = box_y + box_h - i;
            v_pline( vw, 2, line );
        }

        vswr_mode( vw, mode );        /* return to old mode */
        vsl_width( vw, 1);
        vsl_color( vw, 1);
        return(off);
    }
}
```


Program 4-12. label.c

```

label( vw, value, x, y )
int vw;
double value;
int x, y;{

    char str[80];
    int len;

    len = strval( value, str );
    v_gtext( vw, x, y, str );
    return( len );
}

```

Program 4-13. strval.c

```

strval( value, str )
double value;
char *str;{

    char *p;

    sprintf(str,"%f",value);
    for( p = str; *p; p++ )          /* find the end of the string */
        ;
    while( *--p == '0' )              /* trim trailing zeros */
        *p = 0;
    if( *p == '.' )
        *p = 0;
    return( strlen( str ) );
}

```

The strval function uses the C function sprintf to convert a double-precision floating-point value to a character string. It trims off trailing zeros and, if the remainder ends in a decimal point, it is also removed. The string length is found using the C strlen function, and the length is returned to label.

When strval returns the string and its length, label prints the string using the GEM routine v_gtext and returns the length to grid so the colored background can be drawn to the right of the labels.

The grid function can draw either a grid, or simply horizontal lines like those used for bar graphs. If the increment value for the x-axis in the data file is equal to the maximum x value, only horizontal background lines will be drawn. The statement in grid

```
if( x < max_x )
```

checks for this condition. If x is less than max_x, the program draws the vertical lines for the grid, calling label to put numbers under the first and last line. The width of the second label is calculated before it is plotted. The length will be subtracted from the width of the colored background so the label can be right-justified with the background.

Now the background color and pattern are filled by calling the drawbox function, Program 4-14, from grid.

In drawbox, the interior fill mode to PATTERN is set by calling vsf_interior, and the pattern is selected by calling GEM's vsf_style with pattern

number 3. At the top of the file, you can see that 3 is defined as a hatching pattern. This is a light pattern that looks good in color or in monochrome and doesn't interfere with the finer details of our graph.

GEM's `vsf_color` is used to set the fill color to number 3, which is one that hasn't been used yet and also is the last color available on a medium-resolution screen. On a monochrome screen, color 3 is black, but the light pattern will allow us to see the graph lines on it.

To draw a box with rounded corners, giving a finished look to the graph, `v_rbox` is used. This is one of those VDI routines that use the upper left and lower right corners instead of `x`, `y`, `w`, and `h`, so we add `x` to `w` and `y` to `h` to accommodate it.

Before the return to grid, the pattern and color are reset to what they were at startup by the `openvwrk` function in the envelope library.

Back in grid, the drawing of the grid begins with dotted lines. First, the width of the grid lines is set to 1 pixel and the color is set to 2 using the VDI functions `vs_l_width` and `vs_l_color`. On color monitors, the color associated with 2 is whatever the user has selected for the desktop before running the PLOT program. On monochrome systems, any color other than 0 (white) is black; consequently, this program doesn't require any changes to work in monochrome.

Next, the style of the grid lines and the style of their end markers are set. The line is defined to be DOTTED; however, you can change it to one of the other types defined at the top of the grid file, such as dashes or combinations of dashes and dots. The line ends are set to plain SQUARE, although you can achieve some interesting effects if you set them to ARROW.

Before continuing, the writing mode is saved as it was originally set by GEM to "REPLACE" and the mode is changed to transparent. If the writing mode was left as REPLACE, the grid lines would consist of the line pattern with white between the dots and dashes. This would make the grid lines far too prominent in our graph. Changing the writing mode to TRANSPARENT permits the background to appear behind the patterned lines. The old writing mode is saved because it will be restored when the function is finished. This lets us call grid from functions that expect the default writing mode, since, unlike line width and color, this is not one of the parameters that normally changes.

To get the current writing mode, the `getmode` function, Program 4-15, is called.

`getmode` calls the GEM function `vqt_attributes` to ask for ten items of information about VDI text modes. The program specifically wants to know what the writing mode is set to, which may be REPLACE, TRANSPARENT, XOR, or REVERSE TRANSPARENT. The writing mode is returned to grid.

The old writing mode having been saved, the mode is reset to TRANSPARENT with the GEM `vswr_mode` function.

All the information needed is now known; all the variables are set to draw the graph grid and its labels; and we're ready to draw the lines. The program decides if it must draw vertical grid lines, and if so, scale, Program 4-16, is called to convert the graph units (such as dollars) to pixels.

The scale function maps graph units into pixels. It is called whenever we need to make a graph fill the screen. scale is given a value in graph units (for example, dollars), the maximum and minimum values in graph units, and the total height (or width) in pixels of a window. It returns a value in pixels that is the same proportion to the total size as the first value is to the difference between the minimum and maximum. For example, if the window is 100 pixels high, the first value is \$20, and the minimum and maximum values are \$10 and \$100, then scale will return 11 pixels and you will plot a point in the window 11 pixels high to represent the \$20.

After the conversion, v_pline is called to draw the vertical grid lines. The horizontal grid lines are computed and drawn exactly the same way. Before returning to the line_chart function, grid resets the parameters that were changed—the writing mode, line width, and color—and returns the x value of the left edge of the background so that the graph routines will know where to start the graphs.

Program 4-14. drawbox.c

```
# define HOLLOW      0
# define SOLID       1
# define PATTERN     2
# define HATCH       3

draw_box(vw,x,y,w,h)
int vw, x, y, w, h;{

    int corners[4];

    vsf_interior( vw, PATTERN );
    vsf_style( vw, 3 );
    vsf_color( vw, 3 );
    corners[0] = x;
    corners[1] = y;
    corners[2] = x + w;
    corners[3] = y + h;
    v_rfbbox( vw, corners );
    vsf_interior( vw, HOLLOW );
    vsf_style( vw, 0 );
    vsf_color( vw, 0 );
}
```

Program 4-15. getmode.c

```
/*
** Return the current writing mode (Replace, Transparent, XOR, or
** Reverse Transparent)
*/
get_mode(vw)
int vw;{

    struct {
        int text_face;
        int text_color;
```



```
int angle;
int hor_align;
int ver_align;
int write_mode;
int char_wide;
int char_high;
int cell_wide;
int cell_high;
} info;
```

```
vqt_attributes( vw, &info );
return( info.write_mode );
}
```

Program 4-16. scale.c

```
scale( datum, max, min, isize )
double datum, max, min;
int isize;{

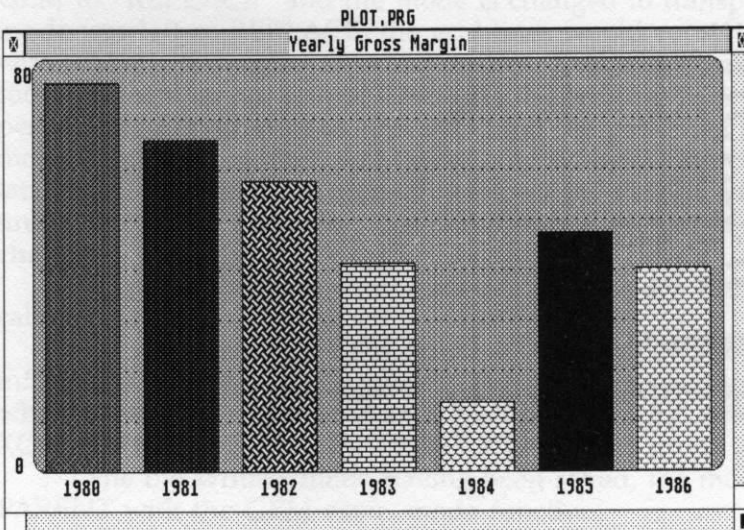
    /**
    **      (datum-min)      =      x
    **      max-min          =      isize
    **
    **/

    return( isize * ((datum-min) / (max-min)) );
}
```

The bar_chart Function: Drawing a Bar Chart

Another very common way to express numerical relationships is with the bar style of chart. Using many of the same subroutines we developed earlier in this chapter, it is a simple matter to create the bar chart in Figure 4-4.

Figure 4-4. A Sample Bar Chart



The data used to create this chart is structured like the data file shown in Program 4-17.

The `bar_chart` function is listed in Program 4-18.

This function begins by saving the value for the grid line increment, just as the `line_chart` subroutine did. There is only one increment value for y , since this is a bar chart and the "grid" will really consist only of horizontal lines.

Also, minimum and maximum values are only required for the y -axis and, since there is only one of each value, we can find them with a simple loop instead of using the range subroutine needed by the line chart program. This loop searches the data to insure that the maximum and minimum values in the data file are correct, and resets them if a larger or smaller value is found.

Skipping over the maximum and minimum values, `bar_chart` calls the `grid` function, explained earlier in this chapter, to put up the background and draw the horizontal lines. The value 0.0 is given for the x -axis to grid so that the vertical lines aren't drawn.

The bars need to be separated by some arbitrary amount of space to keep them distinct from each other. We have chosen to make the space $1/3$ the size of the bars, so we divide the width of the graph box by the number of bars, and then divide by 4 to get the width of a space. The width is then multiplied by 3 to give the width of each bar. All the bars plus their spaces will exactly fill up the width of the graph. Of course, for your version of the program, you can modify the width to be any value you want.

Now each bar and its label is ready to be drawn on the graph.

Program 4-17. `bar.plt`

```
Yearly Gross Margin
BAR
10          # increment
80          # invisible max
0           # invisible min
"1980", 15
"1981", 26
"1982", 34
"1983", 50
"1984", 77
"1985", 44
"1986", 51
```

Program 4-18. `barchart.c`

```
bar_chart( vw, count, data )
int vw, count;
double *data;{

    double max_y, min_y, y;
    int i, bar_space, bar_width, off, loff, ll;
    extern int box_x, box_y, box_w, box_h, gl_hchar, gl_wchar;
    extern char *labels[];

    y = *data++;          /* increment for grid */
    count--;
    max_y = min_y = data[0];
    for( i = 0; i < count; i++ ){
```

CHAPTER 4

```
max_y = max_y < datafil ? datafil : max_y;
min_y = min_y > datafil ? datafil : min_y;
}
data += 2; /* invisible max & min for scaling */
count -= 2;
off = grid( vw, 0.0, y, 0.0, max_y, 0.0, min_y );
bar_space = box_w / count / 4;
bar_width = bar_space * 3;
for( i = 0; i < count; i++ ){
    draw_bar( vw, i * (bar_width + bar_space) + off + bar_space / 2,
              bar_width,
              scale(datafil,max_y,min_y,box_h),
              i+1 );
    ll = strlen( labelsfil ) * gl_wchar;
    if( ll < bar_width )
        loff = box_x + (bar_width - ll) / 2;
    else
        loff = box_x;
    v_gtext( vw,
             i * (bar_width+bar_space) + off + bar_space/2 + loff,
             box_y + box_h + gl_hchar + 1,
             labelsfil );
}
}
```

The draw_bar function. Most of the work for draw_bar, Program 4-19, is done by bar_chart, which calculates its parameters. When draw_bar is called, it's given a location to draw the bar, a bar width and height, and a pattern number for the bar fill. The bar height is given in pixels, calculated by the scale function discussed earlier. The first bar is placed a half-space from the left margin, resulting in another half-space between the last bar and the right margin. This way, the bars stand away from the graph edges for a more pleasing appearance.

draw_bar constructs the corners of the bar from the globally defined background variables, box_x and box_y, and the parameters passed to it.

The draw_bar function sets the interior fill mode to PATTERN and the style to its argument. At the beginning of this file you can see a table of patterns for the index. The patterns have been arranged so that adjacent patterns contrast with each other. The pattern parameter is modulo 24 (the size of the table) so that if there are more than 24 bars, the routine wraps to the beginning of the table.

Color is set to black and then the bar is drawn using the VDI routine v_bar.

As usual, before returning, fill, style, and color are set back to their original state.

Program 4-19. drawbar.c

```
# define HOLLOW      0
# define SOLID       1
# define PATTERN     2
# define HATCH       3

/*
** Put the patterns in a more interesting sequence
```

```

*/
int pat_list[] = {
    1,5,7,16,9,20,8,12,4,19,2,6,3,22,10,13,11,24,14,15,17,18,21,23
};
draw_bar( vw, x, wide, high, pattern )
int vw, x, wide, high, pattern;{

    int xy[4];
    extern box_x, box_y, box_w, box_h;

    xy[0] = box_x + x;
    xy[1] = box_y + box_h;
    xy[2] = box_x + x + wide;
    xy[3] = box_y + box_h - high;

    vsf_interior( vw, PATTERN );
    vsf_style( vw, pat_list[pattern % 24] );
    vsf_color( vw, 1 );

    v_bar( vw, xy );

    vsf_interior( vw, HOLLOW );
    vsf_style( vw, 0 );
    vsf_color( vw, 0 );
}

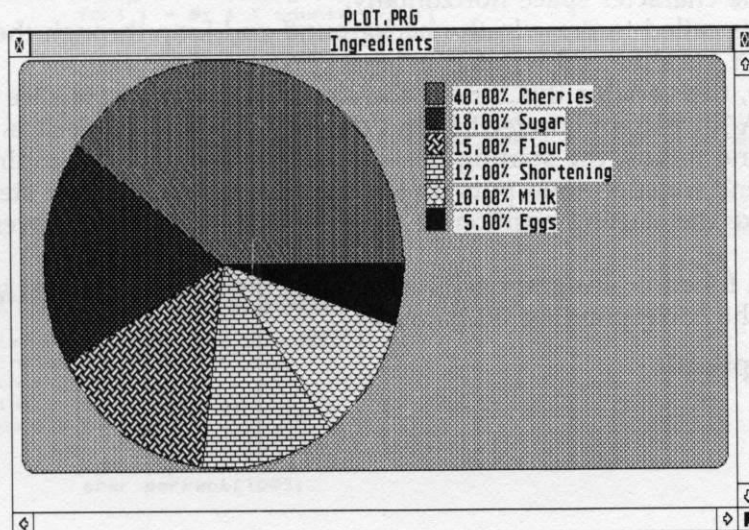
```

Labeling the bars. After the bars are drawn, the labels are centered under them. In `bar_chart`, the length of the label is subtracted, in pixels, from the width of the bar. If it is shorter than the bar, the label is centered by adding half of the difference to the `x` value that is passed to `v_gtext`, which displays the label. If the label is the same length or longer than the bar, it is simply aligned with the left side of the bar.

Drawing Pie Charts

Pie charts, like the one in Figure 4-5, are easy to do using GEM VDI routines.

Figure 4-5. A Sample Pie Chart



The `pie_chart` function simply sums up all the data items, calculates the center point and radius of the pie, then draws the pie slice and labels a legend entry for each data point. The data file for the sample pie chart shown in Figure 4-5 is listed in Program 4-20.

The code for the program to plot the chart is shown in Program 4-21.

The most complicated part about drawing pie charts in windows of varying sizes is making sure that the diameter is smaller than the shortest side of the window. If a radius of 100 pixels on the x -axis is specified, then the radius will be a little less than $1/6$ the screen's width (640 pixels in medium and high resolution). However, the same 100-pixel radius on the y -axis would be $1/2$ the screen's vertical size (in medium and high resolution).

The fact that pixels are taller than they are wide creates the need for some special treatment. GEM uses the x -axis pixels to define the radius. The function must calculate the equivalent number of y -axis pixels. To convert y -axis pixels to x -axis pixels, the function multiplies the number of y -axis pixels by the width of the screen (measured in pixels). Then it divides that result by the height of the screen, also measured in pixels. Now the height and width are in the same units as if the screen were square, and the function can use one-for-one comparisons.

To get the size of the window, `pie_chart` calls the GEM `wind_get` function. If the width of the window is less than the height, then the radius must be $1/3$ the width to keep the pie chart within the window. If the height is less than the width, the radius is made $1/3$ the height.

To position the labels, the starting position of the background box and the diameter of the pie chart are added, adding two more characters of space to offset the labels to the right of the pie.

The center of the pie is in the middle of the background vertically, and the radius plus one character space horizontally.

`draw_box` is called to draw in the background and loop through the data, drawing each pie slice and writing its label.

In the loop, the variable `this_one` is the size of the current pie slice relative to the total. A fill pattern is selected, and the VDI function `v_pieslice` is called with x , y , and radius, the starting position of the slice, and the ending position. The legend function then puts up the pie slice labels. The last step in the loop is to make the starting position of the next slice become the current ending position.

The legend function, Program 4-22, places the pie slice labels next to small patches of the corresponding fill pattern.

Program 4-20. `pie.plt`

```
Ingredients
PIE
"Cherries",    40
"Sugar",       18
"Flour",       15
"Shortening",  12
"Milk",        10
"Eggs",        5
```




Program 4-21. piechart.c

```
# include <gemdefs.h>

# define HOLLOW      0
# define SOLID      1
# define PATTERN     2
# define HATCH      3

pie_chart( vw, count, data )
int vw, count;
double *data;{

    double sum;
    int i, this_one, sofar, x, y, w, h, radius, labx;
    long int max_w, max_h;
    extern int box_x, box_y, box_w, box_h, gl_wbox, gl_hbox, pat_list[];
    extern char *labels[];

    sum = 0.0;
    for( i = 0; i < count; i++ )
        sum += data[i];

    /*
    **      data[i]      =      x
    **      -----      =      -----
    **      sum          360.0
    */
    wind_get( 0, WF_CURRXYWH, &x, &y, &w, &h );
    max_w = box_w;
    max_h = box_h;
    max_h *= w;
    max_h /= h;

    if( max_w < max_h )
        radius = max_w / 3;
    else
        radius = max_h / 3;
    labx = box_x + 2 * radius + gl_wbox * 2;
    x = box_x + radius + gl_wbox;
    y = box_y + box_h / 2;
    sofar = 0.0;
    draw_box( vw, box_x, box_y, box_w, box_h );
    for( i = 0; i < count; i++ ){
        this_one = (data[i] * 360.0) / sum;
        vsf_interior( vw, PATTERN );
        vsf_style( vw, pat_list[(i+1) % 24] );
        vsf_color( vw, 1 );
        v_pieslice( vw, x, y, radius, sofar, sofar+this_one );
        legend(vw, labels[i], labx, box_y + (i+1) * gl_hbox, data[i]/sum);
        sofar += this_one;
    }
    vsf_interior( vw, HOLLOW );
    vsf_style( vw, 0 );
    vsf_color( vw, 0 );
}
```

Program 4-22. legend.c

```
legend( vw, str, x, y, ratio )
int vw;
char *str;
int x, y;
double ratio;{

    int xy[4];
    char percent[100];
```

CHAPTER 4

```
extern int gl_wchar, gl_hchar;

ratio *= 100.0;
xy[0] = x;
xy[1] = y;
xy[2] = x + gl_wchar * 2;
xy[3] = y + gl_hchar;
v_bar( vw, xy );
sprintf(percent, "%5.2f%% %s", ratio, str);
v_gtext( vw, x + gl_wchar * 3, y + gl_hchar, percent );
}
```

legend calculates the percentage of each slice by multiplying the pie slice ratio by 100. Then it draws a small box that fills with the current fill pattern. The `sprintf` function is used to print the label into a string—"40.00% Cherries," for example—and put the string onto the screen using GEM's `v_gtext`.

Putting It All Together

At last, with all the functions entered and compiled, it's time to link them together. The arguments for the link batch program are kept in a file called `link.arg`. Program 4-23 is the listing of this file, with the names of all the functions written for this business graphics application in capital letters.

The `linkit.bat` file, Program 4-24, for the PLOT program, listed below, takes most of its arguments from the `link.arg` file. Again, our programs are written for an ST with a hard disk and we keep our tools on disk C: in the folder `\bin`. If your system configuration differs (no hard disk, for instance), you'll need to change the lines in `linkit.bat` accordingly.

As usual, the linked files are output as `a.prg`, which you will probably want to rename to something descriptive like `PLOT.PRG` or `GRAPHICS.TTP`. Remember that by means of the `.TTP` extension, TOS will give users the opportunity to type in the name of the data file to be plotted when they start the application from the desktop.

Program 4-23. `link.arg`

```
a.68k=gemstart.o,main.o,
CONFIG.O,LABEL.O,STRVAL.O,LINECHRT.O,RANGE.O,SCALE.O,
GRID.O,DRAWBOX.O,GETMODE.O,BARCHART.O,DOIT.O,DRAWBAR.O,PIECHART.O,
LEGEND.O,OPENDATA.O,READDATA.O,STRIPCMT.O,EXTRACT.O,SLCTFILE.O,
env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```

Program 4-24. `linkit.bat`

```
c:\bin\link68 [undefined,symbols,command[link.arg]]
c:\bin\relmod a
c:\bin\rm a.68k
c:\bin\wait
```



5 Creating Menus, Dialog Boxes, and Graphics



5 Creating Menus, Dialog Boxes, and Graphics

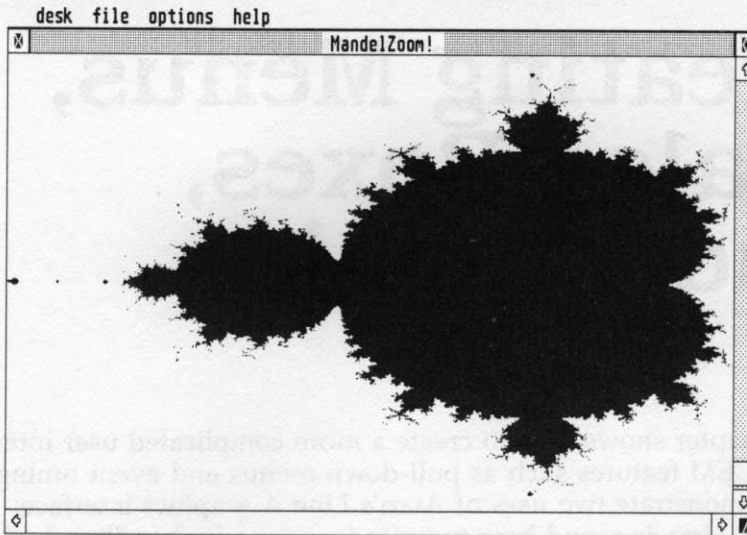
■ This chapter shows how to create a more complicated user interface using GEM features such as pull-down menus and event timing. We also demonstrate two uses of Atari's Line A graphics interface: speeding up screen drawing, and how to write functions for handling keyboard input. The Line A interface resides in ROM.

The program used for explaining the various functions explores a mathematical object called the Mandelbrot set. A Mandelbrot set is an infinite series of numbers that exhibit symmetry, both visual and numerical, as they're plotted. Shown on the Atari screen, the Mandelbrot set produces strikingly beautiful images. There's a practical side to it too. The Mandelbrot set is related to *fractals*, which are mathematical relationships that occupy a "fraction of a dimension"—for instance, somewhere between a line and a plane. Fractal geometry has been used to create background scenes of alien planets in movies and video games.

The complexity of a Mandelbrot set can be seen in Figure 5-1. One distinctive feature of a Mandelbrot-set image is that you can magnify parts of the image and get continually greater detail. The GEM interface, with the mouse and the ability to expand rectangles defined by the user, makes it well-suited to exploring these images.

As in the other application programs in this book, the GEM envelope library from Chapter 2 is used to supply most of the functions for this application. Recall that the intent of the envelope library is to provide a complete set of standard interface routines, requiring only specific connecting routines to be written for each specific application. The list below contains the specific connecting routines for the Mandelbrot program. Each of these functions is explained later in this chapter.

Figure 5-1. A Mandelbrot Set Image



<code>config.c</code>	gives the window a name, program type, and so on
<code>do_menu</code>	determines the item selected from the menu and calls the appropriate sub-routine to handle it
<code>doit</code>	draws the Mandelbrot set
<code>build_tree</code>	sets up the menu tree
<code>mouse_hit</code>	responds to mouse events
<code>got_key</code>	responds to keyboard input
<code>dialog</code>	displays a dialog box for the user to communicate coordinates
<code>just_draw</code>	displays the correct screen

In addition to these connecting functions, we'll also describe the program listings for a number of other functions required for this program to work.

The config.c File

The `config.c` file (Program 5-1) is the one originally described in Chapter 2, but modified to define the basic information required for this particular application.

Very few changes are required to `config.c`. The name is defined as "MandelZoom!" and `i_am_accessory` is set to 0 since this application is not a desk accessory. The Resource Construction Set is not used in this program; `USE_RCS` is undefined and `resource` is set to 0.

Program 5-1. config.c

```
# include <gemdefs.h>

char *wind_name          = " MandelZoom! ";
```

```
# ifdef USE_RCS
char *resource      = "MANDEL.RSC";
# else
char *resource      = 0;
# endif USE_RCS

char *access_name    = " Mandelzoom! ";
int i_am_accessory   = 0;
int sx               = 20; /* small window size */
int sy               = 50;
int sw               = 250;
int sh               = 125;
int slv              = 0; /* small window vertical slider pos */
int slh              = 0; /* small window horizontal slider pos */
int svs              = 1000; /* small window vertical slider size */
int shs              = 1000; /* small window horizontal slider size */
int min_wide         = 100;
int min_high         = 50;
int interval         = 30000;
int events = MU_MESAG ; MU_BUTTON ; MU_KEYBD ; MU_M1 ; MU_M2;
```

The do_menu Function

Chapter 2 discusses the mechanisms in the envelope for handling menus, but does not develop those mechanisms fully. Instead they consist of only the minimum code necessary to include them in the standard envelope library. Program 5-2 develops the critical routines that actually implement menu handling. The routines produce the menu and submenus shown in Figure 5-2. The menu is the horizontal list of items across the top of the window. Vertical submenus, which contain selectable items, appear when the user pulls down the submenu by placing the mouse on a menu item.

Figure 5-2. The Mandelbrot Program's Menu and Submenus

Main Menu:

desk	file	options	help
------	------	---------	------

Submenus:

desk	file	options	help
About MandelZoom	Quit	Coordinates	Controls
Desk Accessory1		Square Box	
Desk Accessory2		Time Drawings	
Desk Accessory3			
Desk Accessory4			

The do_menu function handles menu activation. It relates to the rest of the envelope in the following way. The main function in the envelope calls the multi function which, you may recall, waits for events such as mouse selections or keyboard input. Another event multi waits for is a menu event, indicating that the user has pulled down a menu (we'll explain how this menu got there shortly) and selected a menu item with the mouse.

When a menu event occurs, multi calls the `do_menu` function (Program 5-2), passing it the menu identifier and number of the item in the menu. The identifier and menu item number are established when the menu is built by the `build_tree` function, explained later. The menu identifier in this program (and all other programs in this book) is 0 because only one menu is used and numbering begins with 0. However, the `do_menu` function includes hooks that enable it to handle more than one menu. This helps when you write programs that use multiple menus.

This version of `do_menu` has only one switch case because there is only one menu, and the menu identifier stored in the title variable is always equal to `MAINMENU`. The `do_main_menu` function is called and passes the number of the menu item that was selected. Of course, if there were more than one menu, more switch cases, with their respective function calls, would be included.

When `do_main_menu` has completed its work regarding the selected menu item and has returned, `do_menu` calls the GEM routine `menu_tnormal`. At this point the selected menu item is in reverse video. The `menu_tnormal` changes the selected menu item back into normal video.

At its completion, `do_menu` returns control to multi and the envelope library.

Program 5-2. domenu.c

```
/*
** Handle menu activations.
** Menu messages have two fields of interest to us:
** which menu tree it refers to, and which item in that menu.
** Since all of our menu items are from the main menu at
** the top of the screen, we only handle that case here (MAINMENU)
** although we use a switch to leave a hook for more complicated
** menu structures we might have in the future.
*/

# include <mandel.h>                /* This defines MAINMENU */
                                   /* among other things... */

do_menu(title,item,whand,vw)
int title, item, whand, vw;{

    int ret;
    extern struct object *main_addr;

    ret = 0;
    switch( title ){
    case MAINMENU:
    default:
        ret = do_main_menu(item,whand,vw);
    }
    menu_tnormal(main_addr,title,1);
    menu_tnormal(main_addr,item,1);
    return( ret );
}
```


The do_main_menu Function

Pull-down menus consist of a vertical list of selectable items that appear whenever the mouse pointer touches one of the horizontal menu topics at the top of the screen. All of the vertical selectable items are assigned menu numbers. When the user selects an item, its number is passed to do_menu and then to the correct switch statement for the menu. In this application program, the do_main_menu function handles all the actions the user could request from the menu.

The do_main_menu routine determines exactly what action the user has requested and takes the appropriate action based on the request.

The function checks the menu item number passed to it and compares it to the global variables About, Quit, Coord, Square, Timer, and Ctrl. These variables correspond to selectable menu items and are initialized by the build_tree routine as it builds the menu tree (explained later).

The build_tree function sets up the four top-level menu items shown in Figure 5-3: desk, file, options, and help. The setup_window function in the envelope (explained in Chapter 2) puts this menu on the screen at the top of the window.

GEM also gets into the act—it is GEM that causes the pull-down menus to appear when the user passes the mouse over the top-level menu items. In addition, GEM handles the case when the user pulls down the desk menu and then selects a desk accessory. Except for these instances, the do_main_menu function handles all other menu selections.

The last three lines of this function are executed only if there is a bug in the program. These lines help find typos while debugging the program as you type it in. If you add more selections to the program, this code makes certain you modify do_main_menu to handle any menu items added to build_tree.

Program 5-3. domnmenu.c

```
int make_square = 1;

do_main_menu(item,whand,vw)
int item, whand, vw;{

    char str[256];
    extern struct object *main_addr;
    extern struct object dial_coord[];
    extern int About, Quit, Coord, Square, Timer, Ctrl, do_timit;

    if( item == About ){
        sprintf(str,"[0][%s:%s:%s:%s][ OK ]",
            "Mandelzoom! Quickly draw",
            "the Mandelbrot set as",
            "described in Sci. Amer. Aug",
            "1985. Zoom using the",
            "mouse. Move using arrows."
        );
        form_alert(1,str);
        return(0);
    }
}
```

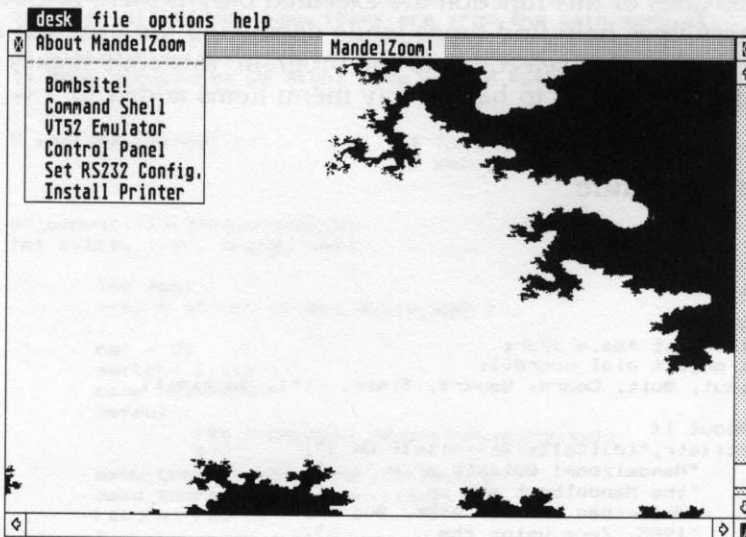
```

else if( item == Quit ){
    return(1);
}
else if( item == Coord ){
    if( coordinates() ){
        clr_display(whand,vw);
        do_display(whand,vw);
    }
    return(0);
}
else if( item == Square ){
    make_square = !make_square;
    menu_ichk(main_addr, Square, make_square);
    return(0);
}
else if( item == Timer ){
    do_timit = !do_timit;
    menu_ichk(main_addr, Timer, do_timit);
    return(0);
}
else if( item == Ctrl ){
    give_help();
    return(0);
}
sprintf(str, "[%s %d] [ OK ]", "Unknown menu number!", item);
form_alert(1, str);
return(0);
}

```

The desk menu. If the user passes the mouse over the desk menu item, the pull-down menu shown in Figure 5-3 appears.

Figure 5-3. The Desk Pull-Down Menu

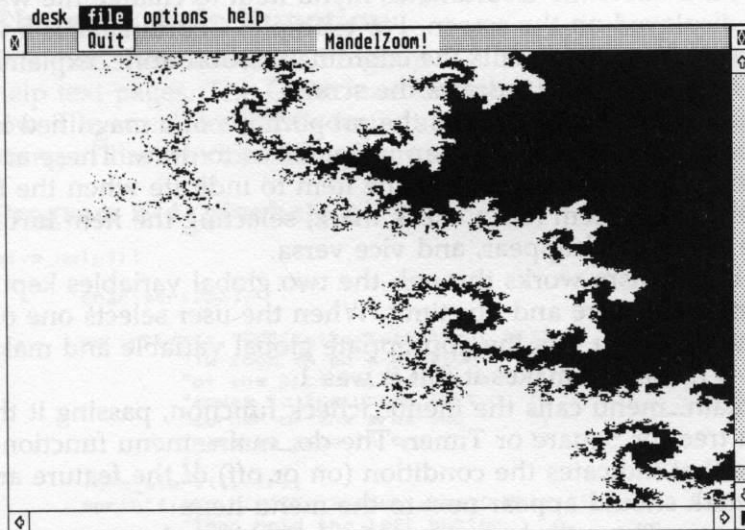


The desk pull-down menu is a special case. GEM lets the programmer specify the names to appear in the first two positions in the menu; in this program, they are *About MandelZoom* and the dotted line under it. Up to six names of desk accessories can follow, which are put in the list by GEM. If there are no desk accessories to pick from, the pull-down menu for desk will end after the dotted line.

If the user selects *About MandelZoom* from the desk menu, GEM sends a menu event to the multi function in the envelope library and it filters down to `do_menu` and then to `do_main_menu`. The item number will be equal to the global variable `About` (initialized in `build_tree`), which signals the program to put a window on the screen with some information about the program. The GEM `form_alert` function is then used to display a window with an OK button and a few terse phrases about the program. Often this message contains the program author's name, copyright notice, and other information about the program or how to use it. The message text for the Mandelbrot program can be seen in the `do_main_menu` listing.

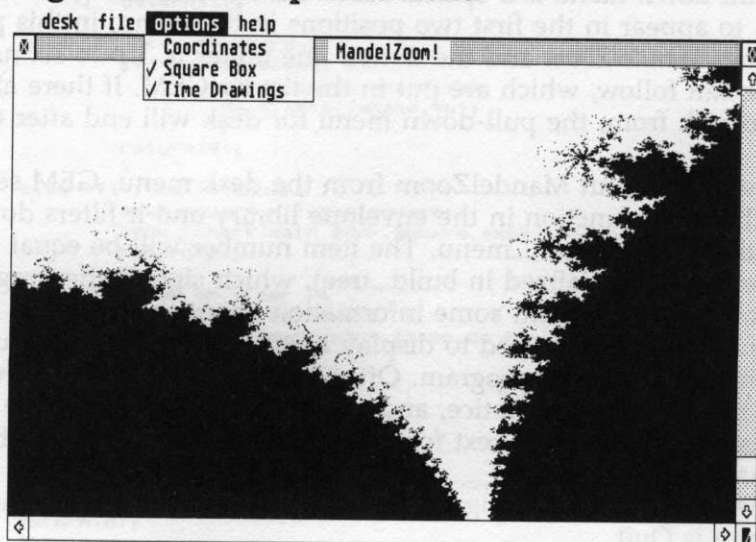
The file menu. As shown in Figure 5-4, one of the selections on the pull-down file menu is *Quit*.

Figure 5-4. The File Menu



Selecting *Quit* causes `do_main_menu` to return 1, defined to mean exit. The `do_menu` function first returns the menu item to normal video to deselect it, and returns the 1 to multi, causing multi and main to exit and return the user to the desktop.

The options menu. The pull-down menu for the top-level options menu item is shown in Figure 5-5.

Figure 5-5. The Options Menu

The user would select the Coordinates menu item to change the way the Mandelbrot set is displayed on the screen. For example, Coordinates is a way to zoom in to a particular area. It calls the coordinates subroutine, explained in detail below, and then clears and redraws the screen.

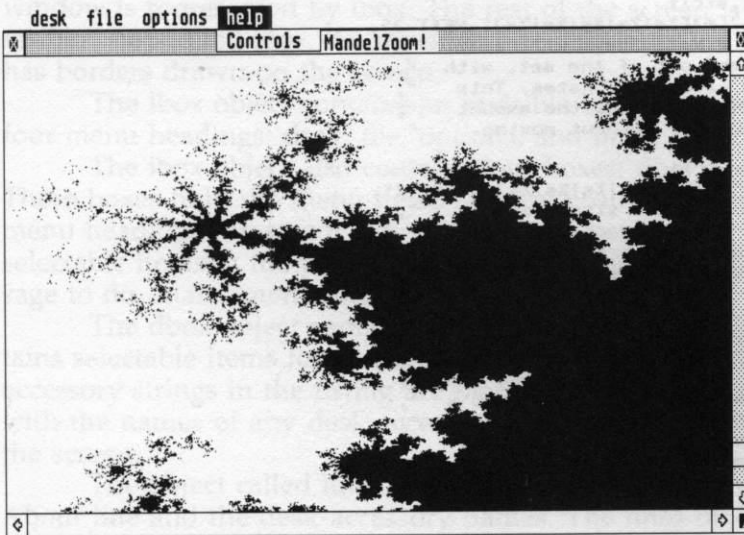
The Square Box menu item adjusts the proportions of a magnified image; Time Drawings keeps track of how long an image takes to draw. These act as toggle switches, with a check mark next to the item to indicate when the feature is on. When the menu item has a check mark, selecting the item turns it off and causes the check to disappear, and vice versa.

The toggle mechanism works through the two global variables kept by `do_main_menu`: `make_square` and `do_timit`. When the user selects one of the menu items, `do_main_menu` sets the appropriate global variable and makes it a 1 (true) if it was 0 (false), or makes it 0 if it was 1.

Then, `do_main_menu` calls the `menu_ichack` function, passing it the index into the menu tree for Square or Timer. The `do_main_menu` function also passes the variable that indicates the condition (on or off) of the feature and whether a check mark should appear next to the menu item.

The help menu. The pull-down menu for help has only one menu item in the program. It is shown in Figure 5-6.

When the user selects Controls, the `give_help` function is called to put up a succession of windows with instructions about the program.

Figure 5-6. The Help Menu

The give_help Function

The `give_help` function, Program 5-4, is very simple, consisting mainly of the help text pages. The GEM function `sprintf` is called to “print” into a string, followed by a call to GEM’s `form_alert` routine, which displays the specified string in the window. (The `form_alert` function is explained in Chapter 2.)

Program 5-4. `givehelp.c`

```
give_help() {
    char str[256];

    sprintf(str, "[0][%s!%s!%s!%s!%s][ NEXT ]",
        "To zoom in on a section",
        "of the picture, move the",
        "mouse to the upper left",
        "corner of the area you",
        "want to zoom in on."
    );
    form_alert(1, str);
    sprintf(str, "[0][%s!%s!%s!%s!%s][ NEXT ]",
        "Then hold the left button",
        "down and drag a rectangle",
        "to the size you want. The",
        "screen will begin drawing",
        "when you let the button go."
    );
    form_alert(1, str);
    sprintf(str, "[0][%s!%s!%s!%s!%s][ NEXT ]",
        "The arrow keys allow you",
        "to see the parts of the",
        "picture above, below, and",
        "to either side. Return"
    );
}
```

```

        "returns you to the start.      "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s!%s!%s!%s!%s][ NEXT ]",
        "The + and - keys will zoom      ",
        "in and out of the set, with      ",
        "the same coordinates. This        ",
        "lets you change the amount         ",
        "of detail without moving.          "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s!%s!%s!%s!%s][ NEXT ]",
        "The vertical slider sets          ",
        "how precise each pixel is          ",
        "to be. High precision is            ",
        "slower but has more detail.         "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s!%s!%s!%s!%s][ LAST ]",
        "The Options menu allows you        ",
        "to see and modify the set's         ",
        "coordinates, square boxes           ",
        "to prevent distortion, and          "
    );
    form_alert(1,str);
}

```

The build_tree Function: Building a Menu Tree

Menus are built in the GEM environment using a data structure called object to define a concept like menu, dialog, icon, box, or button. A menu, then, is a list of objects linked together into a tree. Objects in a tree are said to "contain" other objects, in which case the "contained" object is a child object of the parent object.

Trees consisting of objects are used throughout GEM. They're a device that makes it easy for a programmer to draw subpictures by simply picking a branch of the tree and having GEM draw everything on that branch. Subtrees are also convenient because the position of an object is always given relative to its parent object in the tree. If only the *x* and *y* location of the parent are changed, all the children will be drawn in the new place. The build_tree function, Program 5-5, demonstrates how this data structure is built.

When the menu structures are displayed, the user will see boxes that contain text strings. The box and the text strings are each individual objects. To contain a text string in a box, the size of the box must be the correct width and height for the string object.

To create the boxes that will hold the items in the menu tree, we start with a box that is the size of the whole screen. This box is the root of the tree. The box objects defined are the main parts of the tree, and serve as the parents for all subsequent text strings. This gives us the flexibility to move the menus around just by reorganizing the placement of a box in the tree.

The root box is in the upper left and contains two child boxes (objects),

`lbox` and `ibox`. The long, thin box that contains the menu line at the top of the window is represented by `lbox`. The rest of the screen is contained in an "invisible" box called `ibox`. An invisible box only contains other objects; it never has borders drawn on the screen.

The `lbox` object contains an invisible box called `mbox`, which holds the four menu headings: `desk`, `file`, `options`, and `help`.

The `ibox` object also contains four boxes: `dbox`, `fbox`, `obox`, and `hbox`. These boxes hold the menu items and are positioned below their corresponding menu headings. As a convention, the first letter of each of the indexes of the selectable items in the pull-down menus is capitalized. Also, GEM sends a message to `do_main_menu` anytime the user selects one of the items.

The `dbox` object contains menu items for the desk menu heading. It contains selectable items for `About`, and for up to six desk accessories. The desk-accessory strings in the listing are placeholders. GEM will replace the strings with the names of any desk accessories that exist when the menu is put up on the screen.

The object called `lines` is a string of dashes used to visually separate the `About` line and the desk-accessory names. The `lines` object is "disabled" to make it unselectable and is thus a nonfunctioning menu item.

You can see in the preceding figure what the selectable menu items are for the file menu (`fbox`), options menu (`obox`), and help menu (`hbox`).

The `build_tree` function consists mainly of calls to the `addit` function. Basically here's how `addit` works: The `addit` function is responsible for adding new objects into a tree. The parameters for `addit` are the tree to be added to, the parent object that will contain the new object, a GEM definition that tells the object type, a specification for the object, and the `x`, `y`, `w`, and `h` values for the new object. The `addit` function returns the index of the new object in the tree and maintains the global variable `next_item`, which contains the index of the next unused slot in the tree.

We want to give `addit` the size of the new object in a way that lets us ignore the font resolution (high, medium, or low) until we actually run the program. This is done by using characters instead of pixels to specify the `x`, `y`, `w`, and `h` of the new object. The global variables `Wc` and `Hc` tell `addit` the width and height of one character.

The first call `build_tree` makes to `addit` defines the root object. The second argument to `addit` is always the parent's index, and since root has no parent, `-1` is given for this argument. Note that objects can be bigger than the screen and that we use 80 characters for the width even though only 40 characters are displayed in low resolution.

Before `addit` is called again to add `lbox` under root, then `mbox` under `lbox`, followed by each of the menu-line headers, three pixels are added to the height of a character to make the long box taller than the characters it holds. The three extra pixels are for appearance sake only. After adding all the menu items in the top line, we set the character height back to normal.

Additional calls are made to addit to build most of the remainder of the menu. By studying the code (see below) you can see the parent-child relationships defined by the second argument to addit, and the items returned by addit. Note that, for the object types that use character strings such as G_TITLE, the width argument that is passed to addit is handled by the LEN variable. In addit, the actual length of the character string is computed to give us the width of the box. Thus, we can readily change the character string without worrying about how it may change the width of the box.

A few special features are treated slightly differently. To make the dashed lines unselectable, build_tree is called to set the object state for the lines object to DISABLED directly with the statement

```
t_list[lines].ob_state = DISABLED;
```

The ob_state is one of the characteristics an object can have. Similarly, to indicate that the Square Box and Time Drawings features are turned on by default, the GEM routine menu_ichk is used to put a check mark next to the menu items.

When all the menu items have been added, the LASTOB flag is set as the last object. (The value in next_item always points to the slot after the last object, so next_item - 1 will always point to the last object.)

Finally, build_tree returns the address of the tree to do_main_menu.

Program 5-5. bldtree.c

```
# include <gemdefs.h>
# include <obdefs.h>

# define MAXTREE      64
# define M_BLACK      15L      /* would be 1, but we changed the color map */

# define TRANSPARENT  0
# define THICK        (long)( 0xFFL << 16 )
# define BOXCOLOR      (long)( (M_BLACK << 12) ! (M_BLACK << 8) )
# define BOXTHIN       (long)( BOXCOLOR ! TRANSPARENT ! IP_HOLLOW )
# define BOXBITS       (long)( THICK ! BOXCOLOR ! TRANSPARENT ! IP_HOLLOW )
# define LEN           -2      /* Set the width to the length of the string */

# define xx(item) ((t_list[item].ob_x + t_list[item].ob_width) / Wc)
# define yy(item) ((t_list[item].ob_y + t_list[item].ob_height) / Hc)
# define OFFSET       2 /* so the boxes don't abut the left edge */

int Wc, Hc;
int About, Quit, Coord, Square, Timer, Ctrl;
struct object t_list[MAXTREE];

struct object *
build_tree()

extern int gl_wchar, gl_hchar, next_item;
extern int make_square, do_timit;
int root, mbox, desk, file, options, help;
int dbox, fbox, obox, hbox, ibox, lbox;
int lines, desk1, desk2, desk3, desk4, desk5, desk6;
```


Menus, Dialog Boxes, and Graphics

```

/*
** There are three invisible boxes that hold everything.
** The one called root is the whole screen.
** Root holds a long box called lbox, which holds an invisible
** box called mbox which holds all the menu strings. Lbox and
** mbox are the top line of the screen.
** Root also holds ibox, which is an invisible box that encloses
** the rest of the screen (line 2 to line 25). Ibox holds
** all of the menu items that pop up in boxes under the
** menu strings in the top line. The tree looks like this:
**
** root +---> lbox -----> mbox
**      |
**      +---> ibox
**          +---> dbox
**              +---> About
**              +---> lines
**              +---> desk1
**              +---> desk2
**              +---> desk3
**              +---> desk4
**              +---> desk5
**              +---> desk6
**          +---> fbox
**              +---> Quit
**          +---> obox
**              +---> Coord
**              +---> Square
**              +---> Time Drawings
**          +---> hbox
**              +---> Ctrl1
**
next_item = 0;
Hc = gl_hchar;
Wc = gl_wchar;
root = addit(t_list,-1,G_IBOX,0L,0,0,80,25);

Hc = gl_hchar + 3;
lbox = addit(t_list,root,G_BOX,BOXTHIN,0,0,80,1);

mbox = addit(t_list,lbox,G_IBOX,0L,OFFSET,0,27,1);

desk = addit(t_list,mbox,G_TITLE," desk ", 0, 0,LEN,1);
file = addit(t_list,mbox,G_TITLE," file ", xx(desk), 0,LEN,1);
options = addit(t_list,mbox,G_TITLE," options ",xx(file), 0,LEN,1);
help = addit(t_list,mbox,G_TITLE," help ", xx(options),0,LEN,1);

ibox = addit(t_list,root,G_IBOX,0L,0,1,80,14);
Hc = gl_hchar;
dbox = addit(t_list,ibox,G_BOX,BOXBITS,OFFSET,0,19,8);

About = addit(t_list,dbox,G_STRING," About MandelZoom ",0,0,LEN,1);
lines = addit(t_list,dbox,G_STRING,"-----",0,1,LEN,1);
t_list[lines].ob_state = DISABLED;
desk1 = addit(t_list,dbox,G_STRING," Desk Accessory 1 ",0,2,LEN,1);
desk2 = addit(t_list,dbox,G_STRING," Desk Accessory 2 ",0,3,LEN,1);
desk3 = addit(t_list,dbox,G_STRING," Desk Accessory 3 ",0,4,LEN,1);
desk4 = addit(t_list,dbox,G_STRING," Desk Accessory 4 ",0,5,LEN,1);
desk5 = addit(t_list,dbox,G_STRING," Desk Accessory 5 ",0,6,LEN,1);
desk6 = addit(t_list,dbox,G_STRING," Desk Accessory 6 ",0,7,LEN,1);

fbox = addit(t_list,ibox,G_BOX,BOXBITS,xx(desk)+OFFSET,0,6,1);
Quit = addit(t_list,fbox,G_STRING," Quit ",0,0,LEN,1);

```

```

obox = addit(t_list,ibox,G_BOX,BOXBITS,xx(file)+OFFSET,0,18,3);
Coord = addit(t_list,obox,G_STRING," Coordinates ",0,0,LEN,1);
Square = addit(t_list,obox,G_STRING," Square Box ",0,1,LEN,1);
menu_ichack(t_list,Square,make_square);
Timer = addit(t_list,obox,G_STRING," Time Drawings ",0,2,LEN,1);
menu_ichack(t_list,Timer,do_timit);

hbox = addit(t_list,ibox,G_BOX,BOXBITS,xx(options)+OFFSET,0,11,1);
Ctrl = addit(t_list,hbox,G_STRING," Controls ",0,0,LEN,1);

if( next_item > 0 )
    t_list[next_item - 1].ob_flags != LASTOB;
return( t_list );
}

```

The addit Function

The useful `addit` function, part of the envelope library (Program 2-29), adds items to a tree data structure. In this application, it is used to construct the menu tree; in later programs it will be used to construct complicated dialog boxes that would be difficult to create without the `addit` function.

After checking whether there is room for the new item being added by comparing the `next_item` variable to `MAXTREE`, `addit` checks to see if the width argument is the special value `LEN`. If it is, the statement

```
w = strlen(spec);
```

returns the number of characters passed to `addit` in the `spec` variable. This way, the GEM `strlen` function is allowed to count the characters for us and the character string can be changed without having to change the width.

If the parent has no children yet, this object will head the list of children. Consequently, the parent's `ob_head` field (part of GEM's object definitions) is filled with the index for the current item, stored in `next_item`.

The width and height are compared to the child's and, if they are too small to hold the child, a message is displayed. To keep the message on the screen until a key is typed, the BIOS calls `Bconin`. Then the parent's size is adjusted to be large enough for the child. The reason a message is displayed, instead of silently adjusting the size, is to help you track errors during your programming if the menu is wrong.

Next, `addit` fills in the field definitions for the object being defined. Some of the fields are set to defaults like `NONE` and `NORMAL`, and others are calculated from the `x`, `y`, `w`, and `h` parameters. The links between parents, children, and siblings are set by the GEM routine `objc_add` using the link definition fields `ob_next`, `ob_head`, and `ob_tail`.

After `objc_add` has linked this object into the tree, the index of the object is returned and incremented so it points to the next slot for the next call.

The doit Function: Drawing the Mandelbrot Set

The doit function, Program 5-6, is the heart of the program.

This function contains the mathematics to calculate the Mandelbrot set and the calls to draw the image on the screen. The set is constructed from the deceptively simple statement:

$$Z = Z^2 + C$$

The variable, Z , and the constant, C , are both complex numbers. To plot a Mandelbrot set, the x -axis of the screen is used to represent the "real" component of the complex number and the y -axis to represent the "imaginary" component. For each pixel on the screen, the C value is the x value and y value of the pixel, so the point (35,50) on the screen gives the value $35 + 50i$ for C .

To derive a color value for a pixel the program iterates, calculating $Z = Z^2 + C$ until Z^2 is greater than 4 or until it has looped more than a predefined maximum number of times. The color of the pixel is determined by the number of loops. If the maximum number of loops is made, the color is black; if not, the color is determined by the low bits of the iteration count.

You can see what's involved in calculating the color of each pixel in the image by studying the three nested loops that do the work. The first loop steps through all the rows of pixels: 167 on a color monitor, and 343 on a high resolution, black-and-white monitor. The second loop steps through all the columns of pixels: 615 on a color monitor, and 620 on a high resolution, black-and-white monitor. The innermost loop counts the number of times the complex equation is evaluated, which is variable and can be up to 1000 times in this program. If all the maximum iterations occur, the inner loop can be executed up to 212,660,000 times. Obviously, it needs to be as fast as possible.

The mathematics for the inner loop are one complex multiply, one complex add, one complex compare, and one complex assignment. Complex numbers are usually expressed with floating-point numbers: one for the "real" part, and one for the "imaginary" part. In the program, after some algebraic manipulation, the mathematics are done with three multiplies, four adds, four assignments, and one compare.

The ST has no hardware floating-point support and does all floating-point arithmetic in software. If a floating-point multiply operation takes $1/3$ millisecond and there are 637,980,000 operations, it could take two days just to multiply.

By using integer arithmetic, however, all the multiplies can be done in $9-1/3$ minutes. To represent floating-point numbers using integers, the program multiplies each number by a scaling factor, which must be taken into account whenever the numbers are used. We chose 2^{13} as the scaling factor because the multiplies can be done quickly by shifting. We chose the value 13 because it gives 13 bits to the right of the decimal place and 19 to the left. This is just enough accuracy to plot the set, and leaves a magnification capability of $8192 \times$ for zooming in on a portion of the set.

When the inner loop is finished, the program knows the point to be plotted on the screen. Efficiency is important here also, since there are over 100,000 points to plot. A fast way to plot a point, while still letting TOS worry about things like screen resolution, is to use the Line A Graphics Interface. The Line A Interface is a set of sixteen quick-entry points into the operating system's graphics code. For our program we use only the first two entry points, initializing the Line A code so that we can use it and "put pixel" for drawing the pixel on the screen.

Setting up the Line A interface to work in C calls for some special manipulations that are done at the beginning of the file. The pointer `putpixel` is defined as a pointer to a subroutine. It is set to point to two words of hand-assembled code that we have put in the array `line_a`. This hand-assembled code is a function that enters the operating system at the "put pixel" entry point and then returns.

The same device is used to get into the operating system's "initialize line A" entry point in the routine `line_A_init`. When `line_A_init` is called, it returns a pointer to a block of memory that contains the arguments for the GEM `putpixel` routine. These arguments are the *x* and *y* coordinates and the color of the pixel. Thus, to plot a point, we set `*x` to the *x* coordinate, `*y` to the *y* coordinate, `*color` to the color, and call `putpixel`.

The `doit` function has other functions such as terminating long plots if a key is pressed. We use the `IS_CHAR` macro, which calls TOS directly using the TOS bios routine, to see if a key has been pressed. If there is another character waiting to be read, `GET_CHAR` (another macro that calls bios) is used to read it. Then the bell is rung twice with the statement

```
printf("\7\7");
```

The mouse is displayed, and the program returns to `do_display`, one of the functions in the envelope library, explained in Chapter 2.

The `doit` function also lets the user set the number of iterations for calculating a pixel by moving the vertical slider box. The size of the slider box is set to one character high and is positioned at the bottom of the vertical scroll area at the right edge of the window. Whenever `doit` is called, the GEM `wind_get` routine is used (see Chapter 2) to read the slider box position and set the variable `niter` from it.

The program also tells how long it took to plot the image. To do this some fixed reference point must be established. The program uses this fixed point to figure the starting and ending time from it. For this program, January 1, 1980, is the fixed reference point. The time it takes to plot a set is calculated in the `doit` function by calling the `time_it` function, which returns the number of seconds since 1980. Then the plot is completed. When that occurs `time_it` is called again. The elapsed time in seconds is figured by subtracting the latest value from the first; then `time_print` is called to figure the time in minutes and seconds and to display the elapse time in a window.

The color map is changed to a set of colors in a pleasing gradual scale with the colors function. Just before exiting, the map is restored to its old values in the do_cleanup function.

The last thing doit does before returning to do_display is to save the screen. This will allow just_draw to quickly restore the screen when it is needed, so it won't have to be plotted all over again.

Program 5-6. doit.c

```
#
# define SCALE          8192L
# define INSIDE          2048
# define LSCALE          13
# define LSCALE2         12
# define PLOT_TYPE       long

# include <gemdefs.h>
# include <osbind.h>

/*
 * Mandelzooom
 * as described in Scientific American, August 1985
 * draws beautiful fractal patterns on the screen.
 * Handles monochrome, medium resolution (4 colors) color, and
 * low resolution (16 colors) color.
 *
 * Most implementations of Mandelbrot set generators use
 * floating point arithmetic in the inner loop (where millions
 * of calculations must be performed to generate the picture)
 * and take half an hour to six hours to draw the picture.
 *
 * This program can draw the complete set in less than 2 minutes
 * due to the use of fixed point scaled integer arithmetic
 * in the inner loop. Because of this speed, exploration of the
 * Mandelbrot set at very high resolutions (3000 iterations per
 * pixel or higher) become possible, generating very complex and
 * beautiful displays.
 */

struct lineAinfo {
    int vplanes;
    int vwrap;
    int #contrl;
    int #intin;
    int #ptsin;
    int #intout;
    int #ptsout;
};

typedef struct lineAinfo #info;

/*
 ** Here we build a short subroutine by hand...
 */
static short int line_a[] = {
    0xa001,          /* line A 'put pixel' interrupt */
    0x4e75           /* return from subroutine */
};

static int (#putpixel)() = (int (#)()) line_a;
info i_ptr;
```

CHAPTER 5

```

static info
line_A_init(){

    static short int line_a[] = {
        0xa000,      /* initialize line A code */
        0x4e75      /* return from subroutine */
    };

    static info (*init_A)() = (info (*)()) line_a;

    return( (*init_A)() );
}

#define MAX_NPIXEL      640

# define PRT            0
# define AUX            1
# define CON            2
# define MIDI           3
# define KEY            4

# define IS_CHAR(x)     bios(1,(x))
# define GET_CHAR(x)    bios(2,(x))

double orig_real        = -2.0;
double orig_imag        = -2.0;
double side_r           = 4.0;
double side_i           = 4.0;
int color_mask          = 0xf;
int do_timit            = 1;    /* Print out how long it took to do it */

doit(whand,vw)
int whand, vw;{

    register PLOT_TYPE z_real, z_imag, z2_real, z2_imag;
    register int count;
    register PLOT_TYPE c_real, c_imag;
    register int niter;
    register int i, j;
    int *color, *x, *y;
    double float_r_pixels, float_i_pixels;
    long int t, time_it();
    int wwork, hwork, xwork, ywork;
    static int vertical    = -1;
    int    n_r_pixel;
    int    n_i_pixel;
    double increment[MAX_NPIXEL];
    extern int gl_hchar;

    clr_display(whand,vw);
    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    if( vertical < 0 ){
        vertical = 1000;
        wind_set( whand, WF_VSLIDE, vertical, 0, 0, 0 );
        wind_set( whand, WF_VLSIZE, gl_hchar, &j, &j, &j );
    }
    else {
        wind_get( whand, WF_VSLIDE, &vertical, &j, &j, &j );
    }
    vertical = 1000 - vertical;
    i_ptr = line_A_init();    /* setup for line A graphics calls */
    color = &i_ptr->ptsin[0];
    x = &i_ptr->ptsin[0];
    y = &i_ptr->ptsin[1];
    colors(whand,vw);
    niter = vertical + 16;    /* at least 16 gradations in color */
    hide_mouse();
    float_r_pixels = n_r_pixel = wwork;
    float_i_pixels = n_i_pixel = hwork;
}

```

```

for (j = 0; j < n_r_pixel; j++) /* Precompute increment */
    increment[j] = (orig_real + side_r*j / float_r_pixels) * SCALE;
if( niter < color_mask )
    niter = color_mask;
t = time_it();
for (i = 0; i < n_i_pixel; i++) {
    c_imag = (orig_imag + side_i * i / float_i_pixels) * SCALE;
    for (j = 0; j < n_r_pixel; j++) {
        z_real = c_real = increment[j];
        z_imag = c_imag;
        for (count = 0; count < niter; count++) {
            z2_real = z_real * z_real;
            z2_imag = z_imag * z_imag;
            z2_real >>= LSCALE;
            z2_imag >>= LSCALE;
            if( z2_real + z2_imag > 4 << LSCALE )
                break;
            z_imag = z_real * z_imag;
            z_imag >>= LSCALE2;
            z_real = z2_real - z2_imag;
            z_imag += c_imag;
            z_real += c_real;
        }
        if( count >= niter )
            count = color_mask;
        else {
            count &= color_mask;
            if( count == color_mask )
                count = 0;
        }
        *x = j + xwork;
        *y = i + ywork;
        *color = count;
        (*putpixel)();
    }
    if( IS_CHAR(CON) ){ /* bail out */
        GET_CHAR(CON);
        printf("\7\7");
        show_mouse();
        return;
    }
}
show_mouse();
if( do_timit )
    time_print( time_it() - t, niter, whand );
save_screen(whand);
}

```

The time_it and time_print Functions

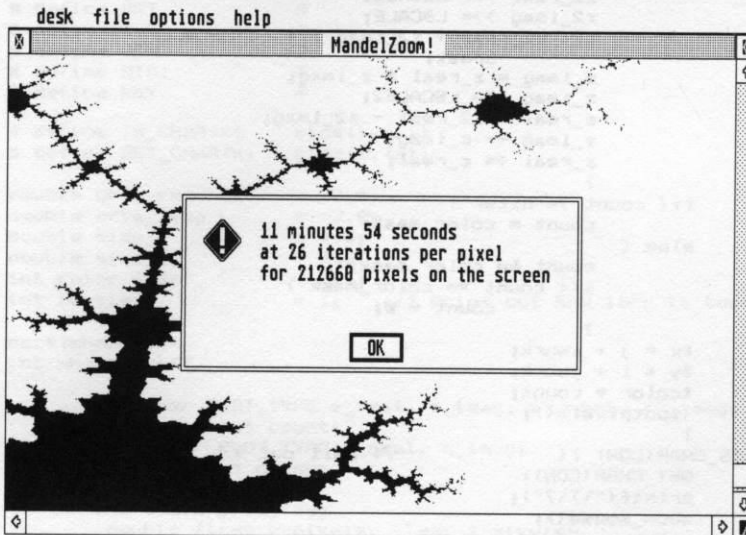
These two functions (Program 5-7) convert the date and time from the way they are stored by GEM into a number that represents the seconds since midnight, January 1, 1980, and displays the elapsed plot time in a window. GEM stores the date and time in two separate integers, packing the year, month, and day in one integer and the hour, minute, and second in another.

The time_it function converts the date and time into seconds in a straightforward manner. The year, month, and day are unpacked and then the number of whole days is calculated by adding the days of each year since 1980, then the days of each month since January, and finally the days since the month began. If this month is after February in a leap year, an additional day is

added. The number of days is converted to seconds as the hour, minute, and second values are added. Note that GEM counts only every other second because there isn't room to store the seconds accurately, so the program multiplies the seconds by two.

The `time_it` function calls the `time_print` routine, which converts the duration in seconds into days, hours, minutes, and seconds, and uses the `GEM_form_alert` function to display them in a window. (See Figure 5-7.)

Figure 5-7. The Window Showing the Elapsed Plot Time



Program 5-7. `timit.c`

```
#include <osbind.h>
#include <obdefs.h>
#include <gemdefs.h>

long int years[] = {
    /* 80 81 82 83 84 85 86 87 88 89 */
    366L, 365L, 365L, 365L, 366L, 365L, 365L, 365L, 366L, 365L,
    365L, 365L, 366L, 365L, 365L, 365L, 366L, 365L, 365L, 365L,
    366L, 365L, 365L, 365L, 366L, 365L, 365L, 365L, 366L, 365L,
    365L, 365L, 365L, 365L, 366L, 365L, 365L, 365L, 366L, 365L,
    366L, 365L, 365L, 365L, 366L, 365L, 365L, 365L, 366L, 365L
};

long int months[] = {
    31L, 28L, 31L, 30L, 31L, 30L, 31L, 31L, 30L, 31L, 30L, 31L
};

long
time_it() {
    long now;
    char str[128];
    int x, date, time, year, month, day, hour, minute, second;

    date = gemdos(0x2a); /* Tgetdate() */
```



```

year = (date >> 9) & 0177;
month = (date >> 5) & 017;
day = date & 037;
time = gemdos(0x2c); /* Tgettext() */
hour = (time >> 11) & 037;
minute = (time >> 5) & 077;
second = time & 037;
now = 0;
for( x = 0; x < year; x++ )
    now += years[x];
now += months[month-1];
now += day-1;
if( years[year] == 366 && month > 2 )
    now++;

/*
** 'now' is the number of days since 1980
*/
now *= 24L;
now += hour; /* hours since 1980 */
now *= 60L;
now += minute; /* minutes since 1980 */
now *= 60L;
now += second * 2; /* seconds since 1980 */

return(now);
}

time_print(secs,niter,whand)
long int secs;
int niter, whand;{

    unsigned int days, hours, minutes, seconds;
    static char d[16], h[16], m[16], s[16], str[80], res[80], pix[80];
    long int calc;
    int xwork, ywork, wwork, hwork;

    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    calc = hwork;
    calc *= wwork;
    sprintf(res,"at %d iterations per pixel",niter);
    sprintf(pix,"for %d pixels on the screen",calc);
    seconds = secs;
    seconds %= 60;
    secs /= 60;
    minutes = secs;
    minutes %= 60;
    secs /= 60;
    hours = secs;
    hours %= 24;
    secs /= 24;
    days = secs;
    d[0] = h[0] = m[0] = s[0] = 0;
    if( days )
        sprintf(d,"%d day%s ",days,days == 1 ? "" : "s");
    if( hours )
        sprintf(h,"%d hour%s ",hours, hours == 1 ? "" : "s");
    if( minutes )
        sprintf(m,"%d minute%s ",minutes, minutes == 1 ? "" : "s");
    if( seconds )
        sprintf(s,"%d second%s",seconds, seconds == 1 ? "" : "s");
    sprintf(str,"%11[%s%s%s%s:%s:%s][ OK ]",d,h,m,s,res,pix);
    form_alert(1,str);
}

```

The colors Function

The colors function, Program 5-8, is called by the doit function to create the color map and get the pixel colors in our image. GEM orders the colors in the color map differently from the Line A graphics we are using. Therefore, we reorganize the colors to correspond to the values Line A uses, going from white to black with a continuous color spectrum between.

The colors routine saves the old colors with the subroutine sav_colors, then chooses new colors according to the screen resolution. To get the screen resolution we call the macro GET_REZ, which we defined as xbios(4).

If rez equals 0, this is a low-resolution monitor with 16 colors available. The colors are set from an array into which the colors are arranged in a continuous spectrum. Since only 16 colors are available, the global variable color_mask is set hexadecimal f (the four lower bits are set to 1) to allow 16 colors. This variable is used by the doit subroutine to convert large numbers into the color range using modula arithmetic.

If rez equals 1 this is a medium-resolution monitor with four colors available, so four colors are chosen from the array and assigned to the first four colors in the map. The global variable color_mask is set to 2-bits (hexadecimal 0x3) to allow four colors.

If rez equals 2, this is a high-resolution monitor with only black and white available. The variable color_mask is set to one 1-bit (hexadecimal 0x1) to allow two colors. Because black-and-white monitors only show the black part of the Mandelbrot set, we zoom in to expand the central black part of the image and make it fill more of the screen. This is done by setting the variables orig_real, orig_imag, side_r, and side_i to smaller values than the initial values they are given in the doit function.

Program 5-8. colors.c

```
# include <osbind.h>
# include <obdefs.h>
# include <gemdefs.h>

# define GET_REZ(x)    xbios(4)
# define SET_REZ(x)    xbios(5,-1L,-1L,(x))

/*
** GEM mixes up the colors, so that 0 is always white, and 1 is always
** black, no matter how many colors the device supports.
** Unfortunately for those who use the Line A graphics, it means that
** to set the colors from GEM and use them from Line A, we must undo
** the mapping. When you tell GEM you want color 1, GEM puts a 15 into
** video RAM. When you tell GEM 6, video RAM gets 3. The following
** table sets up the colors so that we can think in terms of the
** video RAM values that the Line A uses, and use 0 for white, 15 for black,
** and get a continuous (red,yellow,green,blue,violet,red) spectrum in
** between.
*/
struct {
    int red, green, blue;
} cols[16] = {
    {1000, 1000, 1000},    /* 0 white */
    {0, 0, 0}             /* 15 black */
}
```

```

(1000, 0, 0),          /* 1 red */
(1000, 429, 0),        /* 2 red/orange */
(1000, 1000, 0),       /* 4 yellow */
(0, 1000, 0),          /* 6 green */
(1000, 714, 0),        /* 3 yellow/orange */
(571, 1000, 0),        /* 5 yellow/green */
(0, 1000, 714),        /* 7 blue/green */
(0, 1000, 1000),       /* 8 light blue */
(0, 714, 1000),        /* 9 medium blue */
(0, 286, 1000),        /* 10 almost blue */
(714, 0, 1000),        /* 12 purple */
(1000, 0, 571),        /* 14 red/violet */
(0, 0, 1000),          /* 11 definitely blue */
(1000, 0, 1000),       /* 13 violet */
);
colors(whand,vw)
int whand, vw;{

    int x, rez;
    static int did_save = 0;
    extern int color_mask;
    extern double orig_real, orig_imag, side_r, side_i;

    if( did_save == 0 ){
        save_colors(vw);
        did_save = 1;
    }

    rez = GET_REZ(0);
    if( rez == 0 ){
        color_mask = 0xf;
        for( x = 0; x < 16; x++ )
            vs_color(vw,x,&cols[x]);
    }
    else if( rez == 1 ){
        color_mask = 0x3;
        vs_color(vw,0,&cols[0]);
        vs_color(vw,1,&cols[1]);
        vs_color(vw,2,&cols[12]);
        vs_color(vw,3,&cols[14]);
    }
    else {
        if( color_mask != 0x1 ){ /* first time only */
            orig_real = -1.78;
            orig_imag = -1.125;
            side_r = 2.25;
            side_i = 2.25;
        }
        color_mask = 0x1;
        vs_color(vw,0,&cols[0]);
        vs_color(vw,1,&cols[1]);
    }
}

```

The save_colors and rest_colors functions. The save_colors function, Program 5-9, puts all the colors in the GEM color map into an array before the color map for Line A graphics is rearranged. save_colors is called by the colors function, which is called by doit to pick colors for the pixels. save_colors calls the GEM routine vq_color to read the color map and store each color in the array.

The rest_colors function, Program 5-9, restores the colors to the original state they were in when the program started—before the color map was rearranged for Line A graphics in the colors routine. rest_colors is called by the

`do_cleanup` function, which is called by `main` just before the program exits. `rest_colors` calls the GEM routine `vs_color` to read the array saved by `save_colors` and set each color back to its original value.

Program 5-9. `savcolor.c`

```
/*
** Save and restore the color map.
*/

# include <osbind.h>

# define REALIZED      1

struct {
    int red, green, blue;
} old_cols[16];

int old_rez;

save_colors(vw)
int vw;{

    int x;

    old_rez = Getrez();
    for( x = 0; x < 16; x++ ){
        vq_color( vw, x, REALIZED, &old_cols[x] );
    }
}

rest_colors(vw)
int vw;{

    int x;

    Setscreen(-1L,-1L,old_rez);
    for( x = 0; x < 16; x++ ){
        vs_color( vw, x, &old_cols[x] );
    }
}
```

The `do_cleanup` function. Chapter 2 describes a default version of the `do_cleanup` function. It's one of the functions that is usually tailored for each application and called by `main` just before the program exits. For this program, `do_cleanup`, Program 5-10, simply calls the `rest_colors` function to set the color map values back to GEM's values before exiting.

Program 5-10. `doclean.c`

```
do_cleanup(whand,vw)
int whand, vw;{

    rest_colors(vw);
}
```


Looking Around the Mandelbrot Image

The image plotted by a Mandelbrot set is intricate. It can be viewed to greater and greater magnification, revealing seemingly endless details. Explore the set by zooming in and moving around the image with the mouse or the keyboard, and by entering coordinates to see specific image sections. As described below, each of these three methods requires a separate function.

The size, shape, and location of the viewing window into the Mandelbrot image is changed by altering some variables that define a rectangle in the complex plane (recall that the Mandelbrot set is drawn as a complex figure). The altered variables are `side_r`, `side_i`, `orig_real`, and `orig_imag`. They're used in the following functions.

The `mouse_hit` Function

The easiest and most natural way to zoom in on a part of the image is to frame the part in a box. The portion of the image enclosed by the box then expands to fill the entire window, thus magnifying the part.

The user can draw this box with the mouse by pointing to the upper left corner of area to be magnified, and dragging a "rubber rectangle" until the rectangle frames the chosen area. As soon as the user presses the mouse button, GEM sends a message to our application. The multi function receives the message that a mouse event has occurred and calls the `mouse_hit` function to handle it. Program 5-11 is the code for `mouse_hit`.

The `mouse_hit` routine is called when a mouse button is pressed and when it is released, its function being to keep multi informed about the current state of the mouse button. For this application, `mouse_hit` will call a GEM routine that uses the mouse-release message, so that multi doesn't have to do anything until the next time a mouse button is pressed. Only the button press is important; button releases are ignored by returning to multi if the parameter `buttondown` is 0, which indicates the button is up.

When `mouse_hit` is called, it is passed several parameters: the x and y location of the mouse when the button was pressed, the keyboard state (whether the SHIFT, ALT, or CONTROL keys were pressed), the number of clicks on the mouse button, and handles for the window and virtual workstation.

Most of the real work is done in the GEM routine `graf_rubberbox`, which controls the drawing of the rubber box. When `graf_rubberbox` receives the message that the user has released the mouse button, it returns the width and height of the rectangular area of the image that will be expanded to fill the window.

The `kstate` parameter is examined after `graf_rubberbox` returns to see if the SHIFT key was down when the mouse button was pressed and released. If the SHIFT key was down, the program will zoom away from the set, reducing the current window's contents to fit in the area defined by the new rectangle and filling the rest of the new window with the surrounding area. If no SHIFT

key was down, the program zooms into the set, magnifying the area in the defined rectangle to fill the screen.

The scaling algorithm is essentially the same one used in the PLOT program. To zoom out, the program multiplies by the old size and divides by the new. To zoom in, it multiplies by the new size and divides by the old.

The `graf_rubberbox` function sets a flag indicating whether or not it was successful. If it was successful, it receives the message from GEM that the mouse button was released, and returns to `mouse_hit`. If `graf_rubberbox` was not successful, it returns without processing the mouse-button-released message and, therefore, `mouse_hit` must inform multi that it is still expecting the "mouse up" message. The `butdown` variable is set to the appropriate state so it can be returned and multi will know which button state to wait for.

The global variable `make_square` is checked to see if the user set it by selecting it from the menu. If it was set, then the width and height of the new rectangle are averaged to make a square. This prevents the image from being distorted when the new rectangle is not the same shape as the screen.

Finally, the GEM `graf_growbox` function is called to draw a growing box for feedback to the user, the screen is drawn, and the button's new state is returned to the multi function.

Program 5-11. mousehit.c

```
# include <osbind.h>
# include <obdefs.h>
# include <gemdefs.h>

# define ANY_SHIFT      3
# define XOR_MODE       3
# define RPLC_MODE      1

# define MIN_WIDE       25
# define MIN_HIGH       25

mouse_hit(butdown,x,y,kstate,num_clicks,whand,vw)
int butdown, x, y, kstate, num_clicks, whand, vw;{

    double float_r_pixels, float_i_pixels;
    int new_x, new_y, new_h, new_w;
    int xwork, ywork, wwork, hwork;
    extern int make_square;
    extern double orig_real, orig_imag, side_r, side_i;

    if( butdown == 0 )
        return(1);
    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    float_r_pixels = wwork;
    float_i_pixels = hwork;
    if( graf_rubberbox( x, y, MIN_WIDE, MIN_HIGH, &new_w, &new_h ) ){
        x -= xwork;
        y -= ywork;
        if( kstate & ANY_SHIFT ){
            orig_real = orig_real + side_r*float_r_pixels/x;
            orig_imag = orig_imag + side_i*float_i_pixels/y;
            side_i = side_i * float_i_pixels / new_h;
            side_r = side_r * float_r_pixels / new_w;
        }
    }
    else {
```

```

orig_real = orig_real + side_r*x/float_r_pixels;
orig_imag = orig_imag + side_i*y/float_i_pixels;
side_i = side_i * new_h / float_i_pixels;
side_r = side_r * new_w / float_r_pixels;
}
butdown = 1;          /* rubberbox ate the 'mouse-up' */
}
else
    butdown = 0;          /* evnt_multi must eat 'mouse-up' */
if( make_square )
    side_r = side_i = (side_r + side_i) / 2.0;
graf_growbox(x,y,new_w,new_h,xwork,ywork,wwork,hwork);
do_display(whand,vw);
return(butdown);
}

```

The got_key Function

Whenever the user presses a key, the multi function from the envelope library calls the got_key function, Program 5-12, which must be tailored for each application. For the Mandelbrot-set application, the user could move around in the set with the arrow keys using the got_key function to respond with the proper action.

The got_key function for this application examines the key code in a switch statement. If a switch statement changes any of the variables orig_real, orig_imag, side_r, or side_i, then the program breaks out of the switch and redraws the screen before returning.

When an arrow key is pressed, it increments or decrements the appropriate starting point (orig_real or orig_imag) by the width or length of the plot and the screen is moved by a full page in the direction of the arrow that was pressed.

If the key was a + or a -, got_key zooms in or out by a factor of 2, squaring the picture if the global variable make_square is turned on.

If CONTROL-C was pressed, got_key displays a window asking if the user really wants to exit. The show_form function, discussed earlier, is called to put up a window with two selectable buttons in it: OK and CANCEL. If the user clicks on the OK button, 0 is returned to indicate "exit" to the multi function. If the user selects CANCEL, 1 is returned to cause the keystroke to be ignored.

When the key pressed is the RETURN key, got_key calls show_form again to put up a window with buttons asking if the user wants to return to the original view of the set. If OK is selected, the rectangle is set back to its original size and the back_to_first function is called to redraw the screen from a copy saved earlier in the doit function. (This is explained in the discussion of the just_draw function.)

If the pressed key is not one of those just listed, then a window is displayed containing the message that the character is being ignored.

Program 5-12. gotkey.c

```

/*
** Here is where we handle keystrokes.
** We spend a lot of time with the arrow keys, and let most other keys
** pass on through to do_keys().
** This routine is one of the application specific routines that will
** change a lot depending on the application. A word processor would
** do a lot here, for example.
*/

# define UP__ARROW      0x4800
# define DN__ARROW      0x5000
# define LF__ARROW      0x4b00
# define RT__ARROW      0x4d00
# define C_RETURN       '\r'
# define ESCAPE         0x1b
# define CTRL_C         0x03

got_key(ch,whand,vw)
int ch, whand, vw;{

    extern double orig_real, orig_imag, side_r, side_i;
    extern int make_square;

    switch( ch ){
    case UP__ARROW:
        orig_imag -= side_i;
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
        break;
    case DN__ARROW:
        orig_imag += side_i;
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
        break;
    case RT__ARROW:
        orig_real += side_r;
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
        break;
    case LF__ARROW:
        orig_real -= side_r;
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
        break;
    default:
        switch(ch & 0xff){
        char str[80];
        case '+':
            orig_real -= side_r / 2.0;
            orig_imag -= side_i / 2.0;
            side_r *= 2.0;
            side_i *= 2.0;
            if( make_square )
                side_r = side_i = (side_r + side_i) / 2.0;
            break;
        case '-':
            orig_real += side_r / 4.0;
            orig_imag += side_i / 4.0;
            side_r /= 2.0;
            side_i /= 2.0;
            if( make_square )
                side_r = side_i = (side_r + side_i) / 2.0;
            break;
        case CTRL_C:
            if( show_form("Exit this program") )

```



```

        return(0);
    return(1);
case C_RETURN:
    if( show_form("Return to whole set") )
        return(0);
    orig_real = -2.0;
    orig_imag = -2.0;
    side_r    = 4.0;
    side_i    = 4.0;
    back_to_first(whand,vw);
    return(0);
default:
    do_keys(ch);
    return(0);
}

clr_display(whand,vw);
do_display(whand,vw);
return(0);
}

do_keys(ch)
int ch;{

    char str[64];

    sprintf(str,"Ignoring character 0x%x\n",ch);
    show_form(str);
}

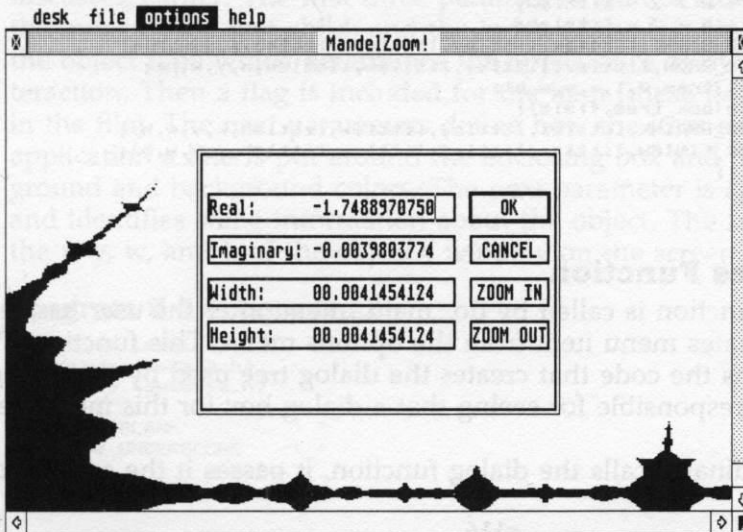
```

The dialog Function

The third way a user can change the view of a Mandelbrot image is by typing in the coordinates of the area to view. Since the exact coordinates are easy to communicate, this is particularly useful for users who want to explain exactly which coordinates to view.

A dialog box like the one shown in Figure 5-8 is used to let the user enter data.

Figure 5-8. The Dialog Box for Entering Coordinates



The dialog function, Program 5-13, handles all user interaction with this box, putting up the dialog window, drawing the expanding and shrinking boxes, and accepting input.

A dialog box is a tree of objects, just like a menu. The coordinates function, which is the next discussed below, calls the dialog function and passes it a dialog tree in the array `box_tree`. The dialog function calls the GEM function `form_center` to create a rectangle that centers the dialog on the screen. Then it calls GEM's `form_dial` with the parameter `FMD_GROW` to start the process of displaying the dialog box and to cause an expanding box to be drawn.

Other GEM routines are called for the dialog box: `objc_draw` puts the dialog box on the screen and `form_do` handles all the mouse and keyboard input. `FMD_FINISH` is a parameter to `form_dial` that frees up memory allocated by `FMD_START`. When that's all finished, `FMD_SHRINK` is called to draw a shrinking box.

When the program returns to coordinates, it passes the index in the object tree of the button the user has selected to end the dialog box session.

Program 5-13. dialog.c

```
# include <obdefs.h>
# include <osbind.h>
# include <gemdefs.h>

dialog(box_tree,field)
struct object *box_tree;
int field;{

    int x, y, w, h;
    int littlex, littley, littlew, littleh;
    int ret;

    if( field < 0 )                /* Atari doc is wrong */
        field = 0;               /* -1 blows up, should be 0 or valid */
    form_center(box_tree,&x,&y,&w,&h);
    littlew = littleh = 50;
    littlex = x + w / 2 - littlew;
    littley = y + h / 2 - littleh;
    form_dial(FMD_START,littlex,littley,littlew,littleh,x,y,w,h);
    form_dial(FMD_GROW,littlex,littley,littlew,littleh,x,y,w,h);
    objc_draw(box_tree,0,1,x,y,w,h);
    ret = form_do(box_tree,field);
    form_dial(FMD_SHRINK,littlex,littley,littlew,littleh,x,y,w,h);
    form_dial(FMD_FINISH,littlex,littley,littlew,littleh,x,y,w,h);
    return( ret );
}
```

The coordinates Function

The coordinates function is called by `do_main_menu` after the user has selected the Coordinates menu item from the options menu. This function, Program 5-14, contains the code that creates the dialog tree used by the dialog subroutine and is responsible for seeing that a dialog box for this menu item appears.

When coordinates calls the dialog function, it passes it the address of the

dialog tree that is defined by coordinates in the array `dial_coord`. This is an example of how to build a tree entirely by hand. It can be done here because the tree is very simple, with only one parent object (a box) that contains all of the child objects. The child objects are the four buttons OK, CANCEL, ZOOM IN, and ZOOM OUT, and four editable text fields where the user will type the coordinates.

The four buttons have been marked as EXIT buttons via flags defined and used in the data structure for the dialog tree. This will cause the `form_do` function to return when the user clicks on a button. The flags `TDEXIT`, `DEFEXIT`, `EXITLAST`, and `TCHEXIT` are defined to represent the exit value and then included as the fifth parameter in the object definitions for each of the buttons.

Editable text fields are special GEM structures that tell the `form_do` function how the data should be entered and displayed. A `TEDINFO` structure (Text EDit INFormation) contains three strings and some information about them such as font, length, and the characteristics of the box that contains the text field. The `form_do` uses the first string as the value to display and let the user change. It uses the second string to label the first string. Wherever an underscore appears in the second string (the *template* string), `form_do` replaces it with a letter from the first string (the *text* string). Thus, the template string "How many:____" and the text string "15" would combine on the screen to read "How many:15__". The third string is the *validation* string, which defines what types of information the user is allowed to type in the field. If an "X" appears in a character position, than any character is allowed. If "9" appears, only a digit can be entered in the position. We have specified one character of any type, allowing a minus sign as the first character, followed by 11 digits.

The dialog tree is placed in the `diag_coord` array. The data structure for each item in the object library is just like the menu in the `build_tree` routine discussed earlier. The first three parameters are the index of the next sibling, the index of the first child, and the index of the last child; they are followed by the object type which determines the appearance of the object and the user interaction. Then a flag is included for the item (these flags were defined earlier in the file). The next parameters define how the item should be drawn; for this application a line is put around the enclosing box and uses the normal foreground and background colors. The next parameter is specific to the object type and identifies some information about the object. The last four parameters are the x, y, w, and h of the object's window on the screen.

Program 5-14. `coordin.c`

```
# include <osbind.h>
# include <obdefs.h>
# include <gemdefs.h>

# define BLANK      ' '
# define UNDERSCORE '_'
```

CHAPTER 5

```

TEDINFO ted_real = {
    "Real:      ",
    "X999999999999",
    IBM, 0, TE_LEFT, 0xFF80, 0, -2, 12, 26
};
TEDINFO ted_imag = {
    "Imaginary: ",
    "X999999999999",
    IBM, 0, TE_LEFT, 0xFF80, 0, -2, 12, 26
};
TEDINFO ted_wide = {
    "Width:      ",
    "X999999999999",
    IBM, 0, TE_LEFT, 0xFF80, 0, -2, 12, 26
};
TEDINFO ted_high = {
    "Height:     ",
    "X999999999999",
    IBM, 0, TE_LEFT, 0xFF80, 0, -2, 12, 26
};

# define TDEXIT          ( EXIT          : TOUCHEXIT : DEFAULT )
# define DEFEXIT         ( EXIT          : DEFAULT    )
# define EXITLAST        ( EXIT          : LASTOB     )
# define TCHEXIT         ( EXIT          : TOUCHEXIT  )
# define EDITLAST        ( EDITABLE     : LASTOB     )

# define B_OK            1
# define B_CANCEL        2
# define B_ZOOM_IN       3
# define B_ZOOM_OUT      4
# define EDIT_FIELD      5

struct object dial_coord[] = {
    -1, 1, 8, G_BOX, NONE, OUTLINED, 0x2FF00L, 0, 0, 38, 12,
    2, -1, -1, G_BUTTON, TDEXIT, NORMAL, "OK", 29, 2, 8, 1,
    3, -1, -1, G_BUTTON, TCHEXIT, NORMAL, "CANCEL", 29, 4, 8, 1,
    4, -1, -1, G_BUTTON, TCHEXIT, NORMAL, "ZOOM IN", 29, 6, 8, 1,
    5, -1, -1, G_BUTTON, TCHEXIT, NORMAL, "ZOOM OUT", 29, 8, 8, 1,
    6, -1, -1, G_FBOXTEXT, EDITABLE, NORMAL, &ted_real, 1, 2, 26, 1,
    7, -1, -1, G_FBOXTEXT, EDITABLE, NORMAL, &ted_imag, 1, 4, 26, 1,
    8, -1, -1, G_FBOXTEXT, EDITABLE, NORMAL, &ted_wide, 1, 6, 26, 1,
    0, -1, -1, G_FBOXTEXT, EDITLAST, NORMAL, &ted_high, 1, 8, 26, 1,
};

double
get_val(str)
char *str;{
    char hold[80], *p;
    double atof();

    while( *str == BLANK || *str == UNDERSCORE )
        str++;
    for( p = str; *p; p++)
        if( *p == BLANK || *p == UNDERSCORE )
            *p = '\0';
    hold[0] = str[0];
    hold[1] = str[1];
    hold[2] = '.';
    strcpy(&hold[3], &str[2]);
    return(atof(hold));
}

```


Menus, Dialog Boxes, and Graphics

```

/*
** Our edit field puts the decimal point in for us.
** It wants to see a string of numbers only, possibly preceded by a minus.
** This routine takes a double, converts it to a string without any
** decimal point, and plugs it into the editable field. This also has the
** nice effect of putting really wild values into our somewhat limited
** range ( between -9.999999... and 99.999999... )
*/
set_val(str,val)
char *str;
double val:(

    char hold[64], *p;

    /* some example values as they change */
    /* -99.5, -1, -.01, 5, 34, 123 */

    if( val < 0 ){
        *str++ = '-';
        val = -val;
    }
    /* 99.5, 1, .01, *, **, *** */
    else if( val < 10 )
        *str++ = '0';
    /* ***** , *, ***, 5, **, *** */
    if( val < 1 )
        *str++ = '0';
    /* ***** , *, .01, *, **, *** */
    sprintf(hold,"%f",val);
    /* 99.5, 1, .01, 5, 34, 123 */
    for( p = hold; *p; p++ )
        if( *p != '.' )
            *str++ = *p;

    /* We return:
    /* It becomes:
    }
    -995, -1, -.001, 05, 34, 123 */
    -9.95, -1.0, -.0.01, 05.0, 34.0, 12.3 */

coordinates()(

    double r, i, w, h;
    char str[80];
    int ret;
    extern double orig_real, orig_imag, side_r, side_i;
    extern int make_square;

    fix_tree(dial_coord);
    set_val(ted_real.te_ptext,orig_real);
    set_val(ted_imag.te_ptext,orig_imag);
    set_val(ted_wide.te_ptext,side_r);
    set_val(ted_high.te_ptext,side_i);

    ret = dialog(dial_coord,EDIT_FIELD);
    if( ret == B_CANCEL ){
        dial_coord[B_CANCEL].ob_state &= ~SELECTED;
        return(0);
    }
    else if( ret == B_ZOOM_IN ){
        dial_coord[B_ZOOM_IN].ob_state &= ~SELECTED;
        orig_real += side_r / 4.0;
        orig_imag += side_i / 4.0;
        side_r /= 2.0;
        side_i /= 2.0;
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
    }
    else if( ret == B_ZOOM_OUT ){
        dial_coord[B_ZOOM_OUT].ob_state &= ~SELECTED;
        orig_real -= side_r / 2.0;
        orig_imag -= side_i / 2.0;
        side_r *= 2.0;
        side_i *= 2.0;
        if( make_square )

```

```

        side_r = side_i = (side_r + side_i) / 2.0;
    }
    else if( ret == B_OK ){
        dial_coord[B_OK].ob_state &= ~SELECTED;
        orig_real = get_val(ted_real.te_ptext);
        orig_imag = get_val(ted_imag.te_ptext);
        side_r = get_val(ted_wide.te_ptext);
        side_i = get_val(ted_high.te_ptext);
        if( make_square )
            side_r = side_i = (side_r + side_i) / 2.0;
    }
    else {
        show_form("Dialog error...");
    }
    return(1);
}
fix_tree(t)
struct object *t;

static int already = 0;
extern int gl_wchar, gl_hchar;

if( already )
    return;
for(;;){
    t->ob_x *= gl_wchar;
    t->ob_y *= gl_hchar;
    t->ob_width *= gl_wchar;
    t->ob_height *= gl_hchar;
    t->ob_height += 2;
    if( t->ob_flags & LASTOB )
        break;
    t++;
}
already = 1;
}

```

The get_val and set_val functions. get_val and set_val set the values in the text strings and read the values back when the form_do function returns. These functions would be simple, except that the numbers in the text string are stored without any decimal points and may contain blanks or underscores. get_val strips out any blanks and underscores, and inserts a decimal point before it calls the GEM atof routine to convert the string to a floating-point number. The set_val subroutine converts the floating-point number back into a text string.

After everything is defined, coordinates has set up the four editable text fields and called dialog, and dialog has returned with its values, coordinates checks the results. If the user has selected the CANCEL button, coordinates deselects the button by returning it to normal video, and returns. If ZOOM IN or ZOOM OUT has been selected, coordinates makes the respective changes in the zoom window coordinates, deselects the button, and returns. And if the OK button has been selected, coordinates uses get_val to convert the text strings, which the user may have modified, and sets the zoom window variables.

The just_draw, save_screen, and copy_first Functions

These three subroutines save the screen images. The user can then always return to the original image, or the screen can be redrawn after a dialog box or accessory window disappears, without having to recalculate the Mandelbrot set. Two images are saved: the first is the original screen; the second is a copy of the screen the last time it was redrawn. By saving the latest screen into a memory buffer, it's a simple matter to copy from the buffer back onto the screen whenever a portion of the screen needs redrawing.

The only difficult part to redrawing after a menu or accessory window has closed is copying (from the memory buffer) only that part of the screen that has been obliterated. The resolution of the screen must also be taken into account because the rectangles are in pixels, which vary with the resolution.

The just_draw, save_screen, and copy_first (Program 5-15) functions work together, so they are kept in one file. They're closely related to the other function, back_to_first—also included in this file—which is discussed in the next section.

The first section of code prepares things for saving the screen by describing the screen's 32,000 bytes as a union of three arrays to handle the three resolutions: low, medium, and high. Since it's a union, all three arrays describe the same 32,000 bytes, organized differently. So 4 bytes can be moved once the arrays are declared as long int arrays. Then, to save the screen, the long ints are copied from the screen into memory in a for loop, using the pointers p and q. The current size of the window (not the screen) is also saved because the program needs to know what portion of the screen belongs to this application when the screen is redrawn.

Every time the screen, or a portion of it, is redrawn, doit calls save_screen to save a copy of it. save_screen hides the mouse so it won't appear in the copy (since it will probably be in a different location when the copy is used) and then calls getlogBase to get the address of the screen. At the beginning of the file getlogBase is defined to be xbios(3).

If this is the first time the screen is drawn for the Mandelbrot application, a copy of the initial screen also needs to be saved in case the user requests the original screen be restored. The variable not_yet is defined as true (1) until the first time save_screen is called, when it is set to false. If not_yet is true, copy_first is called to place a copy of the current screen in the array first_screen.

The just_draw function checks not_yet to see if save_screen was ever called and to see if the size of the window to be redrawn is larger than the last-saved window. If it is larger, then the portion of the window to be redrawn was not saved and doit must be called to recalculate the set.

Because of their outlines, dialog boxes are really larger than the window size that GEM passes to the program. To compensate for this problem, six pixels are added to make sure everything is redrawn properly.

CHAPTER 5

With the use of pointers, tight loops, and long integers, the screen update appears to happen instantaneously.

Program 5-15. justdraw.c

```
#include <osbind.h>
#include <gemdefs.h>

int not_yet = 1;
int savex, savey, savew, saveh;

union u_screen {
    long int low_res[200][400];
    long int med_res[200][400];
    long int high_res[400][200];
} u_screen, first_screen;

/*
** Get the address of the screen
*/
union u_screen *
getlogBase() {
    return( (union u_screen *) xbios(3) );
}

#define MONOCHROME 2
#define MEDIUM_RES 1
#define LOWEST_RES 0

just_draw(whand,x,y,w,h,vw)
int whand,x,y,w,h,vw; {
    register union u_screen *scrn;
    register long int *p, *q, *end;
    register int i, j;

    hide_mouse();
    if( not_yet || x < savex || y < savey || w > savew || h > saveh ) {
        doit(whand,vw);
        show_mouse();
        return;
    }
    if( x > 6 ) /* Adjust for OUTLINED boxes */
        x -= 6;
    if( y > 6 ) /* which are bigger than the xywh */
        y -= 6;
    h += y + 12; /* that we are given by GEM */
    w += x + 12;
    scrn = getlogBase();
    switch( Getrez() ) {
        case MONOCHROME:
            for( i = y; i < h; i++ ) {
                p = &scrn->high_res[i][x >> 5];
                end = &scrn->high_res[i][w >> 5];
                q = &u_screen.high_res[i][x >> 5];
                do {
                    *p++ = *q++;
                } while( p <= end );
            }
            break;
        case MEDIUM_RES:
            for( i = y; i < h; i++ ) {
                p = &scrn->med_res[i][x >> 4];
                end = &scrn->med_res[i][w >> 4];
                q = &u_screen.med_res[i][x >> 4];
            }
    }
}
```



```

        do {
            *p++ = *q++;
        } while( p <= end );
    }
    break;
case LOWEST_RES:
    for( i = y; i < h; i++ ) {
        p = &scrn->low_res[i][x >> 3];
        end = &scrn->low_res[i][w >> 3];
        q = &u_screen.low_res[i][x >> 3];
        do {
            *p++ = *q++;
        } while( p <= end );
    }
    break;
    break;
}
show_mouse();
}

save_screen(whand)
int whand;{

    long int *p, *q, *endpic;
    union u_screen *scrn;
    int xwork, ywork, wwork, hwork;

    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    hide_mouse();
    scrn = getlogBase();
    q = &scrn->low_res[0][0];
    endpic = &u_screen.low_res[200][0];
    for( p = &u_screen.low_res[0][0]; p < endpic; )
        *p++ = *q++;
    show_mouse();
    savex = xwork;
    savey = ywork;
    savew = wwork;
    saveh = hwork;
    if( not_yet )
        copy_first();
    not_yet = 0;
}

copy_first(){

    long int *p, *q, *endpic;

    p = &first_screen.low_res[0][0];
    endpic = &u_screen.low_res[200][0];
    for( q = &u_screen.low_res[0][0]; q < endpic; )
        *p++ = *q++;
}

back_to_first(whand,vw)
int whand, vw;{

    long int *p, *q, *endpic;
    int wwork, hwork, xwork, ywork;

    wind_get( whand, WF_WORKXYWH, &xwork, &ywork, &wwork, &hwork );
    q = &first_screen.low_res[0][0];
    endpic = &u_screen.low_res[200][0];
    for( p = &u_screen.low_res[0][0]; p < endpic; )
        *p++ = *q++;
    just_draw(whand,xwork,ywork,wwork,hwork,vw);
}

```

The back_to_first Function

This routine is called by `got_key` if the user presses the RETURN key and confirms that he or she wants to return to the first screen.

The code for `back_to_first` is the last function in Program 5-14. It simply redraws the screen using the screen saved earlier in the `first_screen` array.

Header Files

The following two header files (Programs 5-16 and 5-17) are also needed to complete our source code.

Program 5-16. `mandel.h`

```
# define MAINMENU      0
```

Program 5-17. `mandefs.h`

```
/*
** Define where the menu items are in the menu array
*/

# define ABOUT_M        9
# define QUIT_M         18
# define COORD_M        20
# define SQUARE_M      21
# define CNTLF_M        23
```

Building the Mandelbrot Program

As with the other programs earlier in this book, we construct the `linkit.bat` batch file for our program as required by the *Atari ST Software Developer's Kit*. If you're using some other version of C, refer to your *User's Manual* for information on linking the necessary files.

`linkit.bat` reads the list of files to be linked from `link.arg`, Program 5-19.

After these two files are constructed, the program is linked by clicking on `batch.ttp` on the desktop and giving `linkit` as the argument. The last thing the `linkit` program does before waiting for the user to press a key is to name the executable file from `a.prg` to `mandlzum.prg`.

Program 5-18. `linkit.bat`

```
c:\bin\link68 [undefined,symbols,command[link.arg]]
c:\bin\relmod a
c:\bin\rmdir a.68k
c:\bin\wait
```

Program 5-19. `link.arg`

```
a.68k=gemstart.o,main.o,
COLORS.O,COORDIN.O,DIALOG.O,GOTKEY.O,DOCLEAN.O,
DOMENU.O,MOUSEHIT.O,SAVCOLOR.O,TIMIT.O,CONFIG.O,
BLDTREE.O,DOIT.O,JUSTDRAW.O,DOMNMENU.O,GIVEHELP.O,
env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```



6 Building a Command Shell Desk Accessory



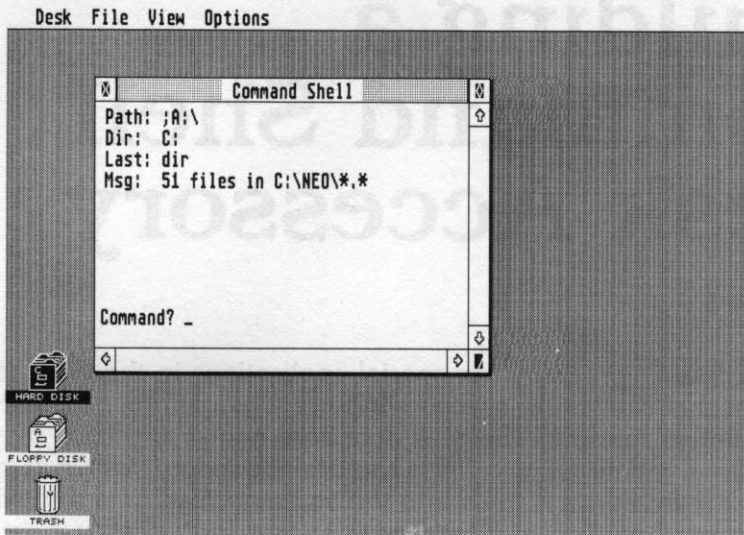
6 Building a Command Shell Desk Accessory

Desk-accessory programs are special applications in the GEM environment that can run in special windows while another regular application is running. The desk-accessory names are listed by GEM in the Desk pull-down menu. GEM collects the desk-accessory names at system boot time, putting the first ones it finds into the accessory list, until a maximum of six accessories is reached. For an accessory to appear in the list, it must be on the boot disk. The limit of six accessories can be restrictive, especially since there are dozens of good desk accessories available.

In this chapter you'll see how to provide many commands in one accessory, thus allowing you to get more mileage out of a single desk accessory. Also, as the accessory program is built, you will see how to construct a file so the same source files can be used for both a desk accessory and a regular program just by changing the way the files are linked. Chapter 7 will explain what you need to do to link the files in this chapter to create a regular application program.

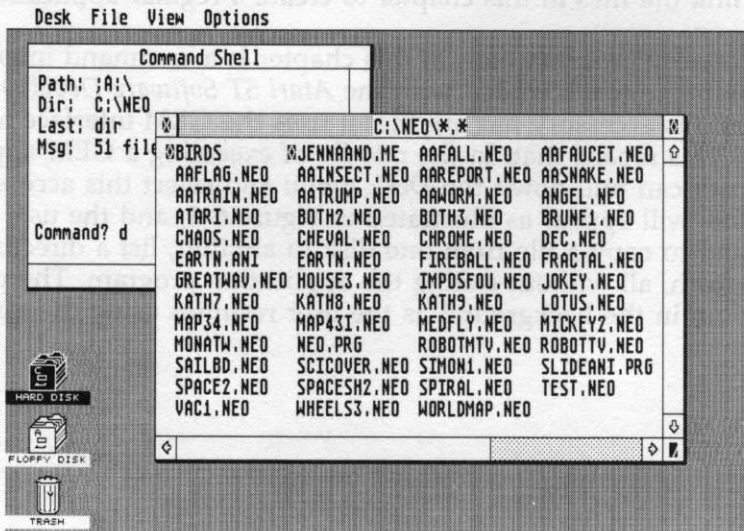
The desk-accessory program built in this chapter is a command interpreter similar to *command.tos*, included with the *Atari ST Software Developer's Kit*. Two advantages this accessory has are that it uses the GEM interface and it is a desk accessory. This means that, in the middle of executing a GEM application program, the user can pull down the Desk menu and select this accessory. An accessory window will appear as illustrated in Figure 6-1, and the user can then type a command to copy a file from one disk to another, list a directory, print a file, and so forth, all without exiting the application program. The printing will actually go on in the background as the user resumes using the application program.

Figure 6-1. The Desk-Accessory Window for the Command.Shell Program



This Command Shell program will be able to execute these commands: copy, move, remove, print, list (print with filename), dir, and chdir. The arguments to these commands can contain pathnames and wildcards, which `dir_list` (see below) expands to a list of full filenames.

Figure 6-2. The `dir` command opens a second window that can be scrolled, resized, moved, and closed.



For some commands, like `dir`, the Command Shell program opens another window on the desktop, as shown in Figure 6-2.

The configac.c File

As with the other programs in this book, this desk-accessory program uses the library of envelope routines developed in Chapter 2 for handling most of the GEM interface programming requirements. To customize the envelope for this application's needs, `config.c` must be adapted. Note that we decided to rename the file to `configac.c` (Program 6-1) because this is the configuration file for the accessory version of the command interpreter; this will help us to distinguish it from the regular application version in Chapter 7.

To set up a program as a desk accessory, the variable `i_am_accessory` is set to 1 and the program is linked as an accessory. The linking process is described in the section "Building the Desk-Accessory Program" near the end of this chapter.

The name "Command Shell" is entered for both `wind_name` and `access_name`, and since this is a desk accessory, only the `access_name` variable will be used. RCS isn't used in this program, so `USE_RCS` is undefined; however, the resource variable is set to `SHELL.RSC` as an example if you want to modify the program.

Program 6-1. configac.c

```
# include <gemdefs.h>

char *wind_name          = " Command Shell ";

#ifdef USE_RCS
char *resource           = "SHELL.RSC";
#else
char *resource           = 0;
#endif USE_RCS

char *access_name        = " Command Shell ";
int i_am_accessory       = 1;
int sx                   = 20; /* small window size */
int sy                   = 20;
int sw                   = 300;
int sh                   = 160;
int slv                  = 0; /* small window vertical slider pos */
int slh                  = 0; /* small window horizontal slider pos */
int svh                  = 1000; /* small window vertical slider size */
int shs                  = 1000; /* small window horizontal slider size */
int min_wide             = 100;
int min_high             = 50;
int interval             = 0;
int events = MU_MESAG | MU_KEYBD;
```

The doit.c and justdraw.c Functions

Clearing and redrawing all or part of the screen are handled by the `doit` and `just_draw` functions (Program 6-2). In previous programs these routines were in separate files because they were fairly complex. Since they're short and closely related in this accessory, it's logical to put them in one file.

The `doit` function clears the screen and calls the `got_key` function, passing it a `-1` parameter so that the screen will be redrawn. `got_key` is explained shortly, but notice that in this program `got_key` does the redrawing because it's the only function which knows about certain data structures; there is some information, such as the pointer into the current command string, which must be hidden from other functions. The C language provides this capability to hide data to prevent the data from being modified in more than one place, making it easier to avoid bugs.

The `just_draw` function is similar to `doit`, except that it only clears the part of the window in the current clipping rectangle. The algorithm used to determine the clipping rectangles is explained at length in the discussion of the `do_redraw` function in Chapter 2.

Program 6-2. `doit.c`

```
doit(whand,vw)
int whand, vw;{

    hide_mouse();
    clr_display(whand,vw);
    got_key(-1,whand,vw);
    show_mouse();
}

just_draw(whand,x,y,w,h,vw)
int whand, x, y, w, h, vw;{

    hide_mouse();
    just_clear(whand,vw);
    got_key(-1,whand,vw);
    show_mouse();
}

open_data(file)
char *file;{

    return(1);
}
```

The `got_key` Function: Responding to Input Commands

Most of the work in the desk accessory occurs in the `got_key` function, Program 6-3. `got_key` handles input from either of two windows: the Shell window and a directory window. From `got_key`, some of the screen-handling functions such as `multi` and `do_redraw` are used in indirect recursion to refresh the accessory window after commands are entered, refresh the directory subwindow that's created when the `dir` command is typed, and collect the characters the user types to initiate commands.

The first two *if* statements in this function are used to determine whether a directory subwindow is open, and whether it should be refreshed or closed. If a window is moved, resized, or uncovered, it must be redrawn. These statements are easier to understand in context, so they'll be discussed in detail with the `do_dir_wind` function.

If the window handle variable (whand) equals `-1`, then the Command Shell window is closed. This should never happen, but this statement has been included as a precaution.

If `got_key` is somehow called when the Shell window is closed, the `got_key` function benignly returns `0` to `multi` so it continues.

If the window handle is not `-1` (and it shouldn't be under normal circumstances), then GEM's `wind_get` function is called to get the size of the Shell window's area and figure out where to put the prompt.

If the character passed to `got_key` in the `ch` variable is `-1`, it means "redraw the window." The contents of the window shown in Figure 6-1 are redrawn.

When the user selects the Command Shell from the Desk pull-down menu, GEM sends the event message `AC_OPEN` to the `multi` function, which calls the `was_message` routine, and the switch statement is entered at the `AC_OPEN` entry point. The `AC_OPEN` event says that a desk accessory has been activated. The Command Shell window appears looking like Figure 6-1. GEM routines have drawn the window frame and the window has all the regular features: It can be sized, full, closed, or moved as the user wishes. The size of this "small" accessory window is dictated in the `configac.c` file.

The text in this window is generated by the `got_key` function, and will be redrawn on the screen whenever `got_key` receives the `-1` argument. In the upper left corner the program prints the current command search path, the current directory, the last command executed, and any status or error message the last command may have issued. The last line is a prompt for a command.

The current command search path consists of the list of directories GEM will search to find a program to run, which TOS sets at boot time. To find out what the path is so it can be printed from the Command Shell window, `got_key` calls the GEM `shel_envrn` routine with the instruction to search for "PATH=" in the Shell's environment space and to put the found value into the array `pathp`. The result is printed on the screen with the GEM functions, `sprintf` and `v_gtext`.

Getting the current directory is done the same way it is in the `PLOT` program, by using `Dgetdrv` to get the drive number and `Dgetpath` to get the directory. The result is printed under the path.

The last command the user types is recorded by `got_key` in the array `lastcom` and the user's command is echoed by printing the characters on the screen beneath the current directory. Because the last command has been recorded, the user is given a shorthand way to repeat the last command. By typing two exclamation points, `!!`, the last command entered will be executed again.

Since some commands result in input or error messages for the user, a way to print the messages on the screen is required. As you'll see in a moment, the array `errmsg` serves as the buffer where commands store their messages.

Next, the location of the last line in the window is calculated for positioning the Command prompt line and the underscore that represents the cursor.

The pointer `p` is set to point into the command buffer and the first character in the buffer is set to 0 to clear it of any previous command and make it ready to receive a new one. `got_key` then returns 0 to the multi subroutine so that it won't exit.

The third function of the `got_key` routine is to respond to the user's keyboard input. The HELP function key must be dealt with separately because it's not an ASCII character, but a 16-bit character whose low 8 bits are zero.

When `got_key` gets the message that the HELP key has been pressed, it calls the `give_help` function, which, except for the text, is very similar to the PLOT program version.

Normal ASCII character keys are sorted out by doing a switch on the low eight bits of the character. If the pressed key is ESCAPE or CONTROL-C, then a 1 will be returned so the multi function will exit back to the desktop. (This statement was added in order to plan ahead to the regular version of this program. Desk accessories can't exit, so ESCAPE and CONTROL-C have no effect from the user's perspective.)

Typing on the keyboard, as long as the character is not RETURN or BACKSPACE, causes the switch to enter the default case. In this case, we hide the mouse, put the character into the buffer using the pointer `p`, print the character on the screen using `v_gtext`, increment the column number, and display the cursor and the mouse.

By pressing RETURN, the user signals that the input command can be executed. `got_key` first clears the `errmsg` array so old messages have no chance of being displayed (in case the command just given doesn't issue a message). To end the string, the command string is terminated with a null. Then the buffer is checked to see if it's empty (`buf[0]`), which means that the user just pressed RETURN or NEWLINE alone. In this case, `got_key` is called recursively with a -1 to redraw the screen, giving the user feedback by causing the screen to blink.

If the buffer contains a command, a check is made to see if it's the special "!!" command, meaning to repeat the previous command. If it is, the previous command is executed by calling the `call_sys` function with the array `lastcom`.

If the command in the buffer is not "!!", the command is copied into the array `lastcom`, and the `call_sys` function is called with the command buffer to execute the command.

A backspace in the buffer means that the user wants to correct a typing error. The pointer `p` is decremented to make it point to the last character that was typed and replace that character with a 0 to terminate the string. To make the characters disappear and the cursor appear to move to the left, a space is printed where the cursor is, the column number is decremented, and the cursor

is printed where the erased character was located. The bell is rung when everything is erased.

The last line in this routine contains a return statement, in which we put a 1 to cause multi to exit the program. In fact, the program should never reach this statement because of the return statements in each switch case. If it does, then modifications to the program have left out a return statement, and by exiting the program, the program alerts a programmer to the fact that a return is missing.

Program 6-3. gotkey.c

```
# include <gemdefs.h>
# include <osbind.h>

# define RETURN          015
# define NEWLINE         012
# define CTRL_C          003
# define ESCAPE          033
# define BACKSPACE       010
# define HELPKEY         0x6200

# define LEFT_EDGE       ( x + gl_wchar * 1 )
# define PATHY           ( y + gl_hchar * 1 )
# define DIRY            ( y + gl_hchar * 2 )
# define LASTY           ( y + gl_hchar * 3 )
# define ERRORY          ( y + gl_hchar * 4 )
# define FILESY          ( y + gl_hchar * 5 )

char errmsg[512];
char *prompt, *pathp;
int dir_window = 0;

got_key(ch,whand,vw)
int ch;
int whand, vw;{

    static char buf[258], curdir[258], lastcom[258], msg[258], *p = buf;
    static int col, line;
    int drv;
    int x, y, w, h;
    extern int gl_wchar, gl_hchar;
    extern int i_am_accessory;

    if( whand == dir_window ){
        if( ch != -1 )
            return(1);
        redo_dir(whand,vw);
        return(0);
    }
    if( whand == -1 )
        return(0);
    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    if( ch == -1 ){
        just_clear(whand,vw);
        hide_mouse();
        prompt = "Command? ";

        shel_envrn(&pathp,"PATH=");
        sprintf(msg,"Path: %s",pathp);
        v_gtext( vw, LEFT_EDGE, PATHY, msg );

        drv = Dgetdrv();
        Dgetpath(curdir,drv+1);
```

CHAPTER 6

```

sprintf(msg,"Dir: %c:%s", drv+'A', curdir );
v_gtext( vw, LEFT_EDGE, DIRY, msg );

sprintf(msg,"Last: %s",lastcom);
v_gtext( vw, LEFT_EDGE, LASTY, msg );

sprintf(msg,"Msg: %s",errmsg);
v_gtext( vw, LEFT_EDGE, ERRORY, msg );

col = x / gl_wchar + 1;
line = (y + h) / gl_hchar - 1;
v_gtext( vw, col * gl_wchar, line * gl_hchar, prompt );
col += strlen(prompt);
v_gtext(vw,col*gl_wchar,line*gl_hchar,"_" );
p = buf;
*p = 0;
show_mouse();
return(0);
}

if( ch == HELPKEY ) (
    give_help(whand,vw);
    return(0);
)

ch &= 0xff;
switch(ch) (
    case CTRL_C:
    case ESCAPE:
        return(1);
    case NEWLINE:
    case RETURN:
        errmsg[0] = 0;
        *p = 0;
        p = buf;
        if( buf[0] ) (
            if( strcmp(buf,"!!!") == 0 ) (
                call_sys(lastcom,whand,vw);
            )
            else {
                strcpy(lastcom,buf);
                call_sys(buf,whand,vw);
            }
        )
        else
            got_key(-1,whand,vw);
        return(0);
    case BACKSPACE:
        hide_mouse();
        if( p > buf ) (
            p--;
            *p = 0;
            v_gtext(vw,col*gl_wchar,line*gl_hchar," " );
            col--;
            v_gtext(vw,col*gl_wchar,line*gl_hchar,"_" );
        )
        else
            printf("\7");
        show_mouse();
        return(0);
    default:
        hide_mouse();
        if( p > &buf[255] )
            printf("\7");
        else (
            *p++ = ch;
            *p = 0;

```



```

        v_gtext(vw,col*gl_wchar,line*gl_hchar,p-1);
        col++;
        v_gtext(vw,col*gl_wchar,line*gl_hchar,"_" );
    }

    show_mouse();
    return(0);
}

return(1);
}

```

The give_help function. In the Command Shell desk-accessory program, the user can ask for help in three ways: pressing the HELP key, or by typing "help" or "?" at the command prompt in the Command Shell window. When the Command Shell runs as a regular program, the user can select Help from the main menu. Any of these events causes got_key to call the give_help function listed in Program 6-4.

Each help screen's text is put into the array str by the GEM sprintf function. The program flips through the six help screens as the user clicks on the NEXT button printed in the help window and returns to got_key after the last screen.

Program 6-4. givehelp.c

```

give_help(whand,vw)
int whand, vw;{

    char str[256];

    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "To copy files, you can type ",
        " copy *.* directory      ",
        " copy *.doc directory      ",
        " or just                    ",
        " copy file1 file2          "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "Move is just like copy, but ",
        "it removes the files after ",
        "it copies them. On single ",
        "file moves, it just renames ",
        "the file.                  "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "To remove files, you can type ",
        " remove *.bad *.old junk.* ",
        " remove ..\\junk\\*.*        ",
        " or just                    ",
        " remove file                  "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "To print files, you can type ",
        " print *.doc *.h *.c         ",
        " print ..\\docs\\*.*          ",
        " or just                    ",
        " print file                   "
    );
}

```

```

form_alert(1,str);
sprintf(str,"[0][%s!%s!%s!%s!%s][ NEXT ]",
        "To show a directory, you type ",
        " dir ",
        " or ",
        " dir *.o *.c *.h ../docs\\*.?",
        "Use chdir to change directory."
);
form_alert(1,str);
sprintf(str,"[0][%s!%s!%s!%s!%s][ LAST ]",
        "Some abbreviations are cp, mv, ",
        "lpr, ls, and cd. The command ",
        "rm is like remove but doesn't ",
        "ask for confirmation. List is ",
        "like print, but with titles. "
);
form_alert(1,str);
}

```

The call_sys, isprg, and set_screen Functions

After the user has typed a command and pressed RETURN, got_key calls the call_sys function (Program 6-5), passing it the buffer containing the command and its arguments. As you look at the code, you'll see that this routine is written so that it can function as a desk accessory, or as a regular program from which other programs can be called.

The GEM function Pexec, which executes another program from the current program, appears in the following code, and will be used by the regular program that's built in Chapter 7. Executing another program does not work from a desk accessory because accessories run as part of the desktop. When a program is executed from an accessory with Pexec, the new program clears the screen and menu using the desktop's resources, but without GEM being aware of it. As a result, the desktop does not know to refresh the screen and menu.

The command buffer is copied into a local array and the command is separated from its arguments by locating the first white space and replacing it with a 0 to terminate the command. The pointer args is left pointing to the first nonblank character after the 0, and str is left pointing to the first character of the command. Then the arguments are placed in the arguments array, reserving the first character for the string length of the arguments, because the Pexec function requires that the first character be the length of the argument string. The first word we isolated is now copied into the command array.

Next, the program must determine if the command that was entered is one of the commands built into this program, or whether another program must be executed. Because accessories cannot execute other programs, we need to know if this is an accessory. This is done by first calling the built_in function to execute the command. If it's not a built-in command, the function will return 0 and then check to see if the current program is a desk accessory. If it is, the program returns, since the desktop will be ruined if an accessory executes another program. If the command was one of the subroutines included in this program, the built_in function will execute it.

If the function finds that this is not a built-in command, and this program is a regular program, then the `call_sys` function calls `find_cmd`, which looks for the command in each possible command directory listed in the `PATH` string found in the environment. If it finds the command, the command's full pathname is returned in the array named `command`, the same array in which the word was originally passed. Because the pathname is returned in this array, 80 bytes have been allocated for the command array, even though the word passed to `find_cmd` will never be that long.

Because executing another command changes the appearance of the screen, it is necessary to save all the information about it in order to restore the screen at the command's completion. It's possible to save the entire screen and then restore it as was done in the `MandelZoom` program; however, there is another way that redraws the screen much faster. It's the method used to achieve the rapid screen changes required by animation applications, and requires changing the pointer to the screen memory location.

GEM keeps two pointers to the screen memory location. These pointers may point to the same place, or to two different places where different screen memories are kept. The screen contents are written to one of these places while the contents of the other screen memory location are displayed by display hardware.

The GEM routines `Physbase` and `Logbase` are used to get the location of the screen's memory, and to save the pointer in `phys` and `log`. The `phys` pointer indicates what the hardware should draw on the screen and the `log` pointer indicates where the screen contents are written. Thus, while the hardware is displaying the contents of one screen memory (pointed to by `phys`), the next screen can be prepared behind the scenes in the location pointed to by `log`. Using the GEM `Setscreen` routine, we can change `phys` to point to `log`, thereby causing a new screen to appear instantaneously, and change `log` to `phys`, to start drawing the next screen. After saving the screen memory's location in `phys` and `log`, we also save the current screen resolution, in case the program being called changes it.

Then the `save_screen` function is called to save the current screen in an array called `screen`, which is aligned to a 256-byte boundary because the screen hardware requires that the screen memory's location begin on an even boundary. This gives us a copy of the original screen, and lets us use GEM's `Setscreen` routine to set the screen location pointer to this location in memory before the command is executed. Executing the command will then write over the copy, but the original screen is saved, with its location stored in `phys` and `log`, and can instantly be restored when it's needed.

Before `Pexec` can be called to execute the command (remember that `Pexec` will call another program only if this is a regular program), the mouse cursor must either be left on or turned off. The `isprg` function checks to see if the command ends in the `.PRG` extension. If it does, it's a GEM program; the mouse cursor should be ON and the text cursor should be OFF, so the

show_mouse function is called to set the cursors. If the command ends in the .TOS or .TTP extensions, then the opposite needs to happen, and GEM's Cursconf is called to display the text cursor and hide_mouse is called to hide the mouse.

With the cursors displayed properly for the command, Setscreen is used to set the screen to the duplicate copy and call Pexec to execute the command. When the command is completed, the cursors are restored to their original state and Setscreen returns the screen pointer to log, its original location. This produces the effect of very quickly restoring the appearance of the Shell window.

The command's exit status is put in errmsg, and the program returns.

Because they're closely related to the call_sys function, the code for isprg and the code for save_screen are included in the same file. The purpose of the short isprg function is to see if the command ends in .PRG. It looks at the last four bytes of a string and returns 1 if they are .prg or .PRG.

The save_screen function in this program is a short version of the same routine in the MandelZoom program. Since save_screen simply saves a copy of the screen, resolution is irrelevant and the data can just be copied as long integers for efficiency.

Program 6-5. callsys.c

```
# include <gemdefs.h>
# include <osbind.h>
# include <obdefs.h>
# include <wfparts.h>

# define LOADGO          0
# define JUSTLOAD        3
# define BASEPAGE        4
# define JUSTGO          5

# define NOCLIP          0
# define CLIP            1

# define EXITAES         0
# define RUNCMD          1

# define LOCK            1
# define UNLOCK          0

# define BLANK           ' '
# define TAB             '\t'

# define SKIPWHITE(x)    while( *x == BLANK || *x == TAB ) x++;
# define SKIPCHARS(x)    while( *x && *x != BLANK && *x != TAB ) x++;

# define HIDE_CURSOR     0
# define SHOW_CURSOR     1

# define ON              1
# define OFF             0

long int screen_buf[215][40];
long int *screen        = &screen_buf[15];

long int ret;
char arguments[256];
```


Building a Command Shell

```
char strbuf[256];
char command[80];

long int
call_sys(str,whand,vw_handle)
char *str;
int whand, vw_handle;{

    register int arglen;
    register char *args, *p;
    int is_graph, is_aes, cx, cy, cw, ch, wx, wy, ww, wh, i, rez;
    int dx, dy, dw, dh, temp[8];
    long int *phys, *log;
    extern int gl_wchar, gl_hchar, i_am_accessory;
    extern struct object *main_addr;
    extern char errmsg[];

    str[255] = 0;
    for( p = strbuf; p < &strbuf[255]; )
        *p++ = 0;
    strcpy(strbuf,str);      /* got_key() will clobber str if called */
    str = strbuf;           /* so we copy it into a safe place */

    SKIPWHITE(str);
    args = str;
    SKIPCHARS(args);
    *args++ = 0;
    SKIPWHITE(args);

    arguments[0] = strlen( args );
    strcpy( &arguments[1], args );

    strcpy(command,str);
    if( built_in(command,args,whand,vw_handle) )
        return;
    if( i_am_accessory )      /* accessories can't handle menus, so */
        return;             /* they can't call programs with menus */
    if( findcmd(command,str) ){
        hide_mouse();
        phys = Physbase();
        log = Logbase();
        rez = Getrez();
        screen &= ~0xff;
        save_screen(screen);
        if( isprg(command) ){
            Setscreen(screen,screen,-1);
            show_mouse();
            i = Pexec(LOADGO,command,arguments,0L);
            hide_mouse();
            Setscreen(log,phys,-1);
        }
        else {
            Cursconf(SHOW_CURSOR,0);
            Setscreen(screen,screen,-1);
            i = Pexec(LOADGO,command,arguments,0L);
            Cursconf(HIDE_CURSOR,0);
            Setscreen(log,phys,-1);
        }
        sprintf(errmsg,"%s returned %d",command,i);
        show_mouse();
    }

    else {
        i = -1;
        sprintf(errmsg,"Can't find '%s'",str);
    }

    return( i );
}
```

CHAPTER 6

```

isprg(str)
char *str;

    int x;

    x = strlen(str);
    if( x > 4 ){
        str = &str[x-4];
        if( strcmp(str, ".prg") == 0 || strcmp(str, ".PRG") == 0 )
            return(1);
    }
    return(0);
}

redo_desk(){

    int x, y, w, h, i, msg[8];

    wind_get( 0, WF_CURRXYWH, &x, &y, &w, &h );
    w += x;
    h += y;
    x = y = 0;
    msg[0] = WM_REDRAW;
    msg[1] = msg[2] = 0;
    msg[4] = x;
    msg[5] = y;
    msg[6] = w;
    msg[7] = h;
    for( i = 0; i < 8; i++ ){
        msg[3] = i;
        appl_write( 0, 16, msg );
    }
}

turn_me(on_off)
int on_off;{

    int msg[8];
    extern int gl_apid, menu_id;

    if( on_off )
        msg[0] = AC_OPEN;
    else
        msg[0] = AC_CLOSE;
    msg[1] = msg[2] = msg[5] = msg[6] = msg[7] = 0;
    msg[3] = menu_id;
    msg[4] = menu_id;
    appl_write( gl_apid, 16, msg );
}

/*
** Get the address of the screen
**/
long int *
getlogBase(){

    return( (long int *) xbios(3) );
}

save_screen(screen)
long int screen[140];{

    long int *p, *q, *endpic;

    hide_mouse();
    q = getlogBase();
    endpic = screen[200];
    for( p = screen; p < endpic; )
        *p++ = *q++;
    show_mouse();
}

```

```
# ifdef DEBUG
debug(str,a,b,c,d,e,f,g)
char *str;
int a,b,c,d,e,f,g;{

    char buf[128], *p;
    static int not_yet;

    if( not_yet == 0 ){
        Rsconf(7,-1,-1,-1,-1,-1);
        for( p = "Hello!\r\n"; *p; p++ )
            Cauxout(*p);
        not_yet++;
    }
    sprintf(buf,str,a,b,c,d,e,f,g);
    for( p = buf; *p; p++ )
        Cauxout(*p);
    Cauxout('\r');
    Cauxout('\n');
}

# endif DEBUG

int Wc, Hc;
struct object d_menu[10];

dummy_up_menu(str)
char *str;{

    extern int next_item, gl_wchar, gl_hchar, Hc, Wc;
    int mbox;

    next_item = 0;
    Hc = gl_hchar;
    Wc = gl_wchar;
    mbox = addit(d_menu,-1,G_IBOX,0L,0,0,80,1);
    addit(d_menu,mbox,G_TITLE,str,0,0,-2,1);
}

menu_on(){

    dummy_up_menu("-----");
    menu_bar(d_menu,ON);
}

menu_off(){

    menu_bar(d_menu,OFF);
}
```

The findcmd.c function. To get the full pathname of a command, the findcmd function, Program 6-6, calls the GEM routine shel_find, which searches each directory in the environment's PATH list, looking for the file.

If shel_find cannot find the command, the user may not have added an extension, so the routine appends the different extensions and searches again. If the command still can't be found, 0 is returned to call_sys, which displays a message telling the user the command couldn't be found.

Program 6-6. findcmd.c

```

findcmd(command, str)
char *command, *str;{

    if( shel_find(command) )
        return(1);
    sprintf(command, "%s.prg", str);
    if( shel_find(command) )
        return(1);
    sprintf(command, "%s.tos", str);
    if( shel_find(command) )
        return(1);
    sprintf(command, "%s.ttp", str);
    if( shel_find(command) )
        return(1);
    return(0);
}

```

The built_in Function

The built-in commands offer quick access to commonly needed functions such as file copying, removing, and printing. They demonstrate another way of entering commands besides the pull-down command menus. The `built_in` function, Program 6-7, handles the following commands for this shell program: `help`, `cd`, `cp`, `mv`, `rm`, `ls`, `print`, and `list`.

For those commands that take filename arguments, the user can enter the `*` wildcard character and the program will expand it into a list of filenames. For example, with the `copy` command the user can enter

cp *.c A: \onefile.c

and, as long as `*.c` is only one file, the command will work. Likewise, the user can type

cp *.c A: \bac*.dir

and the command will work as long as `bac*.dir` is a single directory.

The user can set the current working directory by typing the `cd` or `chdir` commands. The `GEM Dsetpath` routine is called to change to the directory given as an argument.

The `copy` command allows the user to copy a list of files into a directory, in addition to simply copying one file to another. It calls three functions that are explained below, but summarized here. The `copy` command calls `save_last` to save the last filename in the list as the target file or directory. Then, it calls the `dir_list` function to expand any wildcards in the arguments except the last one into a list of filenames. And last, `do_copy` is called to actually copy the files.

The `move` command is similar to the `copy` command, except that the `do_move` function is called instead of `do_copy`.

The `rm` and `remove` commands remove files by calling the `dir_list` function to expand the wildcards, and `do_rm` to remove the files. The `rm` command

will not prompt the user if the file should really be removed, because it passes the DONTASK argument to `do_rm`. The remove command passes the ASK argument to `do_rm`, causing `do_rm` to ask for confirmation.

The `dir` or `ls` commands list the files in a directory. The case where no argument is entered after the command is handled by setting the argument to `.*`, meaning *everything*. To conserve space, the JUSTFILE argument is used when the `dir_list` function is called so that it prints only the filenames, removing any directory path information it finds. Then the `do_dir_window` function is called to display the filenames in a window with the regular GEM interface borders, enabling the user to resize and scroll the window.

The `lpr`, `print`, and `list` commands call the `dir_list` and `print_files` functions to send files to the printer. The `titles` variable is set by the `list` command to cause the filename to be printed at the top of each file. The `list` command causes `title` to be nonzero so that the title is printed; otherwise it is not.

After a command is executed, the screen needs refreshing. All the commands except `dir`, which has its own window, fall through to the `do_display` function call to redraw the screen and print any messages, stored in `errmsg`, that might exist.

If the command is one of the built-in commands, `built_in` returns 1 to let `call_sys` know it can return. When the command is not built in, `built_in` returns 0.

Program 6-7. `builtin.c`

```
# include <gemdefs.h>
# include <osbind.h>
# include <document.h>

# define ASK          1
# define DONT_ASK     0

# define JUSTFILE     0
# define FULLPATH     1

int titles            = 0;

built_in(command,args,whand,vw)
char *command, *args;
int whand, vw;

extern int gl_wchar, gl_hchar, xlines;
extern char errmsg[];
char *last, *save_last();
int ret, x;

ret = 0;
if( command[0] == 0 )
    ret = 1;
else if( strcmp(command,"?") == 0 || strcmp(command,"help") == 0 ){
    give_help(whand,vw);
    ret = 1;
}
else if( strcmp(command,"cd") == 0 || strcmp(command,"chdir") == 0 ){
    Dsetpath(args);
    ret = 1;
}
```

```

else if( strcmp(command,"cp") == 0 || strcmp(command,"copy") == 0 ){
    last = save_last(args);
    if( last ){
        xlines = dir_list(args,FULLPATH);
        do_copy( xlines, last, whand, vw );
    }
    ret = 1;
}
else if( strcmp(command,"mv") == 0 || strcmp(command,"move") == 0 ){
    last = save_last(args);
    if( last ){
        xlines = dir_list(args,FULLPATH);
        do_move( xlines, last, whand, vw );
    }
    ret = 1;
}
else if( strcmp(command,"rm") == 0 ){
    xlines = dir_list(args,FULLPATH);
    do_rm( xlines, DONT_ASK, whand, vw );
    ret = 1;
}
else if( strcmp(command,"remove") == 0 ){
    xlines = dir_list(args,FULLPATH);
    do_rm( xlines, ASK, whand, vw );
    ret = 1;
}
else if( strcmp(command,"ls") == 0 || strcmp(command,"dir") == 0 ){
    if( args[0] == 0 )
        strcpy(args,"*.");
    xlines = dir_list(args,JUSTFILE);
    do_dir_window(xlines,args,whand,vw);
    return(1);
}
else if( strcmp(command,"lpr") == 0 || strcmp(command,"print") == 0 ){
    ret = 1;
    titles = 0;
    if( args[0] == 0 )
        strcpy(errmsg,"Print what?");
    else {
        x = dir_list(args,FULLPATH);
        print_files(x,whand,vw);
    }
}
else if( strcmp(command,"list") == 0 ){
    ret = 1;
    titles = 1;
    if( args[0] == 0 )
        strcpy(errmsg,"List what?");
    else {
        x = dir_list(args,FULLPATH);
        print_files(x,whand,vw);
    }
}
do_display(whand,vw);
return(ret);
}

```

The save_last function. The user can type two types of arguments to the move and copy commands: the source file or files, and the target file or directory. If the target is a file, there can be only one source file to copy into it. If the target is a directory, then a list of files can be moved or copied into it. The program needs to identify the last argument in the list, expand any wildcards, and determine that the argument is a unique file or directory name. It does this with the save_last function, Program 6-8.

To find the last argument, `save_last` goes to the end of the string it was passed, then moves backward one character at a time until it finds a white space. An error message will be displayed if it reaches the beginning of the string without finding a white space, indicating that the user didn't enter a second argument for the target.

The last argument is copied into the `retval` array and passed to the `dir_list` function, which expands wildcards into a list of files and then sorts them alphabetically. We'll talk about `dir_list` in a moment, but for now note that if the target name is not unique, a message is displayed and the program returns. If `dir_list` finds a filename that matches, it returns it in the `dir_strs` array and it is copied into the `retval` array. If a matching name isn't found, then the original string becomes the name of a new file and is copied into `retval`. The white space before the first character is included with the string because the first character distinguishes directories from files. The white space serves as a flag that this is a file. The Shell will create files from arguments, but directories must be created separately.

After the last argument is removed, any extra white space that remains after the last argument is also deleted, and the last argument is returned to the `built_in` function.

Program 6-8. `savelast.c`

```
# define FULLPATH      1
# include <osbind.h>

char *
save_last(str)
char *str;{

    char *p;
    int x;
    static char retval[128];
    extern char errmsg[], *dir_strs[];

    p = str;
    while( *str )                /* go to end */
        str++;
    while( *str != ' ' && *str != '\t' && str > p ) /* find white */
        str--;
    if( str <= p ){
        sprintf(errmsg, "Missing last argument");
        return(0);                /* no white found */
    }
    strcpy( retval, str + 1 );
    if( dir_strs[0] )
        dir_strs[0][0] = 0;
    x = dir_list(retval, FULLPATH);
    if( x > 1 ){
        sprintf(errmsg, "Target is more than 1 file");
        return(0);
    }
    if( dir_strs[0] && dir_strs[0][0] ){
        strcpy( retval, dir_strs[0] );
    }
    else {
```

```

        strcpy( retval, str );
    }
    while( str > p && (*str == ' ' || *str == '\t') )
        *str-- = 0; /* clobber white */
    return( retval );
}

```

The dir_list function: expanding wildcards. When a function needs to expand filenames that contain wildcards, it calls the dir_list function. dir_list takes a list of filenames, wildcards included, and copies the real filenames into an array called storage. As each filename is put in storage, its new address is placed in the array dir_strs. Thus, dir_strs is an array of pointers, each of which points to a filename stored in the storage array. The list of pointers can be manipulated, sorted, or indexed conveniently.

The dir_list file includes the document.h file, shown in Program 6-9.

Program 6-9. document.h

```

# define NFILES          200
# define NLINES          200
# define NCHARS          80

```

The three lines specify the number of files, lines, and characters that dir_list can handle. Keeping these separated makes it easy to make global changes in the sizes.

Two functions that are only used in this routine are included in this file: the cmpfile and next_arg routines. You can see the cmpfile subroutine at the beginning of the listing. This subroutine uses the GEM strcmp subroutine to compare two strings and is used by the GEM qsort subroutine.

The next_arg routine appears at the end of the listing. It finds successive arguments in a list, leaving a pointer pointing to the first character of the argument. It looks for the next argument in a list by finding the white space separator, and then replaces it with a 0 to terminate the string. Remaining white space is skipped and a pointer to the first nonwhite character of the next argument is returned. The pointer points to the end of the string when there are no nonwhite characters.

dir_list, Program 6-10, gets filenames by calling the GEM routines Fsfirst and Fsnext, which return the first filename in a directory, and then succeeding filenames. They return their data to a place called a Data Transfer Address, which must be set by the routine with the GEM Fsetdta subroutine. Because this address is used by other programs, we use Fgetdta to save the old Data Transfer Address and then restore it when we've finished. The Data Transfer Address is set to point to the fs structure. The data returned by Fsfirst and Fsnext is placed in the fs array, whose data structure is defined to correspond to the shape of the returned data. Of importance to this routine is the fs_attr field, in which one of the returned bits indicates whether the name is a directory name.

To expand wildcards in the arguments, two loops are used, one inside the other. The outer one loops to locate each argument and passes it to the inner loop, which looks for files in the directory that match the current argument with its wildcards.

In the outer loop, each argument is isolated by putting a zero in the space between the arguments, and a pointer is moved to successive arguments.

To keep a pathname of an argument, which is indicated because the fullpath variable is nonzero, the get_head subroutine is called to return the path prefix to the filename. The pathname fragment is stored in head, which can then be inserted in front of each filename returned by Ffirst and Fnext to produce a complete pathname that commands like print can use.

Ffirst returns 0 if a file that matches the argument is found. If the file is not one of the special directory names "." or "..", it is copied into storage with a special character called DIR_CHAR that indicates whether it is a directory, and the pathname. On the screen, this special character appears as the box containing a diamond shape. The head variable will be empty if fullpath is 0.

The Fnext function returns the next filename, and the comparing process continues looking for another match. Fnext returns nonzero when there are no more files that match the argument, and the loop terminates.

Ffirst returns nonzero if there are one or no files that match the argument. Any filename that is returned is copied into storage and the pointer in dir_strs is incremented.

At the end of the outer loop, the arguments variable is set to next so the next argument will be used, and the loop repeats. When all arguments have been processed, the old Data Transfer Address is restored and the filenames are sorted alphabetically by calling qsort, the "quicker sort" routine that is part of the C library. The number of files that are found is returned to save_last.

Program 6-10. dirlist.c

```
# include <gemdefs.h>
# include <osbind.h>
# include <document.h>

# define IS_DIRECTORY 16
# define DIR_CHAR 7

# define STORE (8*1024)

char storage[STORE+32], *sp;

char *dir_strs[NFILES];
int dir_index;

cmpfile(a,b)
char **a, **b;{
    return( strcmp( *a, *b ) );
}

dir_list(arguments,fullpath)
char *arguments;
int fullpath;{
```

CHAPTER 6

```

struct fs {
    char fs_junk[21];
    char fs_attr;
    unsigned int fs_time;
    unsigned int fs_date;
    long int fs_size;
    char fs_name[13];
} fs;

long int save_dta;
int x;
char *head, *p, *q, *next, *next_arg(), *get_head();
extern char *dir_strs[NFILES];

sp = storage;
*sp = 0;
dir_strs[0] = sp;
fs.fs_name[0] = 0;
dir_index = 0;
save_dta = Fgetdta();
Fsetdta(&fs);
p = fs.fs_name;
while( arguments[0] && sp < &storage[STORE] ){
    next = next_arg(arguments);
    if( fullpath ){
        head = get_head(arguments);
    }
    else {
        head = "";
    }
    x = Ffirst(arguments, 0x3f);
    if( x == 0 ){
        do {
            if( sp > &storage[STORE] )
                break;
            if( strcmp(p, ".") && strcmp(p, "..") ){
                dir_strs[dir_index++] = sp;
                if( fs.fs_attr & IS_DIRECTORY )
                    *sp++ = DIR_CHAR;
            }
            else
                *sp++ = ' ';
            for( q = head; *sp++ = *q++; ){
                if( sp > &storage[STORE] ){
                    *sp-- = 0;
                    break;
                }
            }
            sp--;
            for( q = p; *sp++ = *q++; ){
                if( sp > &storage[STORE] ){
                    *sp-- = 0;
                    break;
                }
            }
        } while( dir_index < NFILES && Fnext() == 0 );
    }
    else if( p[0] )
        strncpy(dir_strs[dir_index++], p, 13);
    arguments = next;
}
Fsetdta(save_dta);
qsort(dir_strs, dir_index, sizeof(dir_strs[0]), cmpfile);
return(dir_index);
}

char *
next_arg(a)
char *a;

```

```

while( *a && *a != ' ' && *a != '\t' ) /* find white space */
    a++;
if( *a ){
    *a = 0;
    a++;
    while( *a && ( *a == ' ' || *a == '\t' ) )
        a++;
    return(a);
}
else
    return(a);
}

```

The get_head function. With the get_head subroutine, you can identify and save the part of a file's pathname that precedes the filename. This can be especially handy for constructing a complete pathname for a series of filenames, for example, to send them to the printer.

If a pathname of

c:\programs\shell\gethead.c

is given, the get_head function, Program 6-11, will return

c:\programs\shell

The function scans the string for the backslash character and replaces the last one it finds with a 0 to terminate the string. It then returns a pointer to the start of the truncated string.

If there are no backslashes, a null string is returned to the calling routine.

Now let's take a look at the way the individual commands that call dir_list use the data it returns.

Program 6-11. gethead.c

```

char *
get_head(str)
char *str;{

    static char head[2*1024];
    char *h, *last_slash;

    last_slash = 0;
    h = head;
    while( *str ){
        *h++ = *str;
        if( *str++ == '\\' )
            last_slash = h;
    }
    if( last_slash ){
        *last_slash = 0;
        return( head );
    }
    return( "" );
}

```

The `do_copy` Function: Copying Files

The Shell program lets the user copy one or more files into another file or into a directory. It does this by using the `dir_list` function discussed earlier to create a list of filenames which is returned to the `built_in` routine. `built_in` then calls `do_copy` (Program 6-12), passing it the number of files and the target filename (in `last`) into which to copy them. The target can be a file or a directory.

The function checks the `dir_strs` array to verify that there is at least one file to copy, and prints an error message on the screen if there isn't.

To find out if the target is a directory, the first character of the target is checked. Remember that in the `dir_list` routine the `DIR_CHAR` character (a box with a diamond in it) was inserted in the first position of a directory name. The `is_dir` flag was set according to whether or not this was a directory.

For each source file in the global `dir_strs` array, several error-checks are made and then the file is copied to the target. Before copying the source file(s), the program checks that each source file is not a directory. The program also compares the source and target names to make sure they aren't the same. The following paragraphs explain how the program does this checking.

This version of a copy command lets a user copy only one file at a time into another. That is, as a precaution against inadvertent input error, we don't let the user copy several files into one. The `is_dir` variable indicates whether the target is a directory or not. If the target is not a directory, and the number of source files being copied is more than 1, the program prints an error message and returns to `built_in`.

The list of source filenames in the global `dir_strs` array is examined to see if any of the filenames are directories. In this command, if a directory is found among the source files, a message is printed and the function returns to `built_in`, where the user can try retyping the command.

Next, a filename into which the source file will be copied is created. A filename from the `dir_strs` array is copied into the variable `p`, and the first character (which is blank for filenames) is skipped over. If the user has entered a directory as the target, then the program constructs the complete pathname by joining target and `p` (separated by a backslash), and puts the pathname in `tofile`. A plain file simply has its filename put in `tofile`. The name in `tofile` is now compared to the name in `p` to confirm that they are not identical. If they are, an error message is printed and the program returns.

Then the `GEM_fdelete` routine is called to remove any existing target file so we don't run the risk of having parts of an old file appended to the newly copied file, as would happen if the old file were longer than the file being copied. The `Fcreate` is called to create a new file and return its file descriptor in `to_fd`. Next `fopen` is called to open the source file so the data can be read from it.

`sprintf` is called to construct a message about which file is currently being copied, and `do_display` to refresh the screen and display the message. This message feature is especially useful to a user who is copying many files, since the information it contains tells the user the status of the copying process.

Finally, everything has been confirmed and the source and destination files opened. The GEM routines `Fread` and `Fwrite` are called to read from the source file and write to the target. `Fread` returns the number of bytes it has read from the source file and stored in `fbuf`. `Fwrite` collects only that number of bytes from `fbuf`, and writes the data to the new file. `LSIZE` is used because `Fread` needs a 32-bit number for the byte count, and `SIZE` is a 16-bit number. `Fread` returns 0 on end of file, and negative numbers for errors, so the loop will terminate on either condition. When the file is copied, `Fclose` is called to close the source and destination files; then it goes through the loop again for the next filename in the `dir_strs` array.

The ST floppy disks are quite fast, but the software overhead in transferring small records can easily hide that speed. If large records are being transferred, like the 8K records used here, the software is only called upon to set up the transfer once every 8K, and the disk transfer speed is quite good. The constants `SIZE` and `LSIZE` control the size of the transfers. Making them bigger than 8K doesn't increase the speed by much, since we reach diminishing returns when the number of transfers per file gets below 3 or 4. Making them smaller will make the copies take longer, but may be desirable when memory is at a premium, as on 512K STs with a RAMDISK installed.

Program 6-12. docopy.c

```
# include <osbind.h>
# include <document.h>

# define DIR_CHAR      7

# define SIZE           (1024*8)
# define LSIZE          ((long int)SIZE)

do_copy( count, target, whand, vw )
int count;
char *target;
int whand, vw;

    int is_dir, x, to_fd, from_fd;
    long int numbytes;
    char tofile[128], *p;
    extern char errormsg[], *dir_strs[NFILES], fbuf[];

    if( dir_strs[0] == 0 ){
        sprintf(errormsg, "No 'from' file!");
        return;
    }
    if( target[0] == DIR_CHAR )
        is_dir = 1;
    else
        is_dir = 0;
    target++;
    if( is_dir == 0 && count != 1 ){
        sprintf(errormsg, "Usage: copy file file OR copy files directory");
        return;
    }
    for( x = 0; x < count; x++ ){
        p = dir_strs[x];
```

```

p++;
if( is_dir )
    sprintf(tofile,"%s\\%s",target,p);
else
    sprintf(tofile,"%s",target);
if( strcmp(target,p) == 0 ){
    sprintf(errmsg,"Copying '%s' to itself!",target);
    return;
}
Fdelete(tofile);
to_fd = Fcreate(tofile,0);
if( to_fd < 0 ){
    sprintf(errmsg,"Can't create '%s'",tofile);
    return;
}
from_fd = Fopen(p,0);
if( from_fd < 0 ){
    sprintf(errmsg,"Can't open '%s'",p);
    return;
}
sprintf(errmsg,"copy '%s' to '%s'",p,tofile);
do_display(whand,vw);
while( (numbytes = Fread( from_fd, LSIZE, fbuf ) ) > 0 ){
    if( Fwrite( to_fd, numbytes, fbuf ) < 0 ){
        sprintf(errmsg,"Write error on '%s'",tofile);
        return;
    }
}
Fclose(to_fd);
Fclose(from_fd);
}

```

The do_move Subroutine: Renaming Files

The move command is similar to the copy command, except that the source file is removed after the copy. The do_move function, Program 6-13, takes each filename as the source and renames it to the target name by either changing the spelling of the filename, or copying the file and giving it the target name, then deleting the source file. The target can be in a different directory or on another drive.

Most of the do_move function is very similar to the do_copy function discussed in the previous section. After setting up the directory status, path-name (if any), and so on, the program renames the file.

A simple, fast way to change the spelling of a filename to a target name on the same disk and in the same directory is with the GEM Frename routine. Frename checks whether the target is in a different directory and if the target filename exists; if either is true, it returns an error message and the program drops into the copy loop. The program always tries to use Frename to change the spelling before resorting to the more expensive technique of copying the file.

If a filename can't simply be respelled, then the source file must be copied into the new file, with the target filename. This is done in a for loop, which has the same function as the one in do_copy, except that the old file is removed after the copy is completed.

Program 6-13. domove.c

```

#include <osbind.h>
#include <document.h>

#define IS_READONLY 1
#define IS_HIDDEN 2
#define IS_SYSTEM 4
#define IS_VOL_LABEL 8
#define IS_DIRECTORY 16
#define IS_CLOSED 32

#define DIR_CHAR 7

#define SIZE (1024*8)
#define LSIZE ((long int)SIZE)

char fbuf[SIZE];

do_move( count, target, whand, vw )
int count;
char *target;
int whand, vw;{

    int is_dir, x, attributes, to_fd, from_fd;
    long int numbytes;
    char tofile[128], *p;
    extern char errmsg[], *dir_strs[NFILES], fbuf[];

    if( dir_strs[0] == 0 ){
        sprintf(errmsg, "No 'from' file!");
        return;
    }
    if( target[0] == DIR_CHAR )
        is_dir = 1;
    else
        is_dir = 0;
    target++;
    if( is_dir == 0 && count != 1 ){
        sprintf(errmsg, "Usage: move file OR move files directory");
        return;
    }
    if( is_dir == 0 && count == 1 ){
        p = dir_strs[0];
        p++;
        if( Frename( 0, p, target ) >= 0 ){
            sprintf(errmsg, "renamed '%s' to '%s'",
                p, target);
            do_display(whand, vw);
            return;
        }
    }
    for( x = 0; x < count; x++ ){
        p = dir_strs[x];
        p++;
        if( is_dir )
            sprintf(tofile, "%s\\%s", target, p);
        else
            sprintf(tofile, "%s", target);
        if( strcmp(target, p) == 0 ){
            sprintf(errmsg, "Moving '%s' to itself!", target);
            return;
        }
        Fdelete(tofile);
        to_fd = Fcreate(tofile, 0);
    }
}

```

```

if( to_fd < 0 ){
    sprintf(errmsg,"Can't create '%s'",tofile);
    return;
}
from_fd = Fopen(p,0);
if( from_fd < 0 ){
    sprintf(errmsg,"Can't open '%s'",p);
    return;
}
sprintf(errmsg,"move '%s' to '%s'",p,tofile);
do_display(whand,vw);
while( (numbytes = Fread( from_fd, LSIZE, fbuf ) ) > 0 ){
    if( Fwrite( to_fd, numbytes, fbuf ) < 0 ){
        sprintf(errmsg,"Write error on '%s'",tofile);
        return;
    }
}
Fclose(to_fd);
Fclose(from_fd);
Fdelete(p);
}
}

```

The do_rm Subroutine: Deleting Files

To delete files from a disk, the user types the remove command in the shell. The code for removing files is much simpler than for moving them (Program 6-14).

The program verifies that an argument has been entered, and then, if the ask flag is set, asks the user to confirm that the file should be deleted. If the user typed "remove" for the command, then the ask flag is true, and it is false if he typed "rm".

The GEM Fdelete routine takes care of deleting the file from the disk and directory.

Program 6-14. dorm.c

```

#include <osbind.h>
#include <document.h>

do_rm( count, ask, whand, vw )
int count, ask, whand, vw;{

    extern char *dir_strs[NFILES], errmsg[];
    char buf[256], *p;
    int x;

    if( dir_strs[0] == 0 ){
        sprintf(errmsg,"No file to remove!");
        return;
    }
    for( x = 0; x < count; x++ ){
        p = dir_strs[x];
        p++;
        if( ask ){
            sprintf(buf,"Remove '%s'?",p);
            if( show_form(buf) )
                continue;
        }
        sprintf(errmsg,"removing '%s'",p);
    }
}

```



```

do_display(whand, vw);
Fdelete(p);
}

```

The prntfile.c and do_title Functions

When the user gives the shell the print or list command, the function `print_files` (Program 6-15) is called to send files to the printer.

The for loop in `print_files` cycles through the list of files in the `dir_strs` array that was created by `dir_list`, printing the name of each file and a number showing where it is in the list—for example, “main.c:5 of 32”. If a file cannot be opened, an error message is printed on the screen. The list command simply means that the filename should be printed at the top of the file. If the user types list, `built_in` calls `do_title` with a 1 in the titles variable to print the title. Otherwise, `do_title` does nothing.

`print_files` then prints the file by copying from the file into a buffer and, for each character in the buffer, calling the BIOS routine `Cprnout` to send the character to the printer.

If the shell is running as a desk accessory, then printing can occur in the background, while other GEM programs run in the foreground. To allow the other GEM program to run, GEM’s `evnt_timer` routine is called, which GEM uses to decide which program gets to run. GEM can only get control when a GEM call is made, so by calling `evnt_timer` with a 0, GEM can reschedule the different tasks running on the computer.

The programmer has to decide how often to call `evnt_timer`. Calling it for every character will cause very slow printing, even if no other processes are running. Calling `evnt_timer` too seldom will make any competing application run too slowly.

Calling `evnt_timer` every other character is a good tradeoff for printers that operate at 120–180 characters per second. The printer runs almost constantly without slowing down other programs. If your printer has a large RAM buffer or is very fast, you can change the line that keeps track of every odd character:

```
if ( y & 1 )
```

```
to
```

```
if (( y & 3) == 0)
```

to call `evnt_timer` every four characters.

When printing is finished, `print_file` sends a FORMFEED character to advance the paper, and closes the file. The next file in `dir_strs` is opened, and the loop is performed again, until all the files have been printed.

Program 6-15. prntfile.c

```

#include <gemdefs.h>
#include <osbind.h>
#include <document.h>

#define PRINTER 0
#define FORMFEED 014
#define BUFSIZ 4096L

print_files(count,whand,vw)
int count, whand, vw;

{
    int x, fd;
    long int num, y;
    char *fname;
    static char buf[BUFSIZ];
    extern char errmsg[];
    extern char *dir_strs[NFILES];

    for ( x = 0; x < count; x++ ) {
        fname = dir_strs[x];
        fname++;
        sprintf(errmsg,"%s: %d of %d",fname,x+1,count);
        do_display(whand,vw);
        fd = Fopen(fname,0);
        if ( fd < 0 ) {
            sprintf(errmsg,"Can't open '%s'",fname);
            do_display(whand,vw);
            return;
        }
        do_title(fname);
        while ( (num = Fread(fd,BUFSIZ,buf)) > 0 ) {
            for ( y = 0; y < num; y++ ) {
                Cprnout(buf[y]);
                /*
                 ** Every other character, we let GEM decide
                 ** if another program should get a chance.
                 */
                if ( y & 1 )
                    evnt_timer(0,0);
            }
            Cprnout(FORMFEED);
            Fclose(fd);
        }
        do_title(str)
        char *str;

        extern int titles;

        if ( titles == 0 )
            return;
        while ( *str )
            Cprnout(*str++);
        Cprnout('\r');
        Cprnout('\n');
    }
}

```

The `do_dir_window` Function

Issuing the `dir` (directory) command to the shell produces a window on the screen containing a neatly arranged list of filenames like the one in Figure 6-3. The built-in routine calls `do_dir_window`, Program 6-16, to create this window, complete with the sliders, close and resize boxes, and other features of a GEM window. In the code for this program, you'll see how indirect recursion is used to call several of the functions created for the main application, such as `multi` and `got_key`.

First, the function constructs the directory pathname so it can include it at the top of the directory window and in a status message. It uses the GEM routines used before, `Dgetdriv` and `Dgetpath`, to get the drive and path. Then `sprintf` is called to put the pathname in the array, `name`. `sprintf` is called again to construct a message that is printed in the shell window after the directory window is closed, telling the user how many files were in the directory—for example, "32 files in b: \examples \shell". Since the user may have included wildcards in the argument to `dir`, (or given no argument in which case built-in added `*.*` as the default), the status may not be an exact directory name, but something like "14 files in A: \games *.PRG".

Determining the maximum possible size that the directory window can be is the next step. GEM's `wind_get` function, originally introduced in Chapter 2, is used to get the size. After the size is determined, `do_dir_window` calls `calc_dir`, discussed in more detail later, to put all the filenames in columns that will fill a window no bigger than the desktop's work area. In addition, if the list of filenames doesn't fill the window when it's at the maximum size, then `calc_dir` reduces the window to the smallest size it can be and still show all the files.

Now `do_dir_window` sets up the window borders by calling several of the envelope routines developed in Chapter 2. It uses `slide_size` and `slide_pos` to calculate the size and position of the slider boxes, which depend on the size of the window and the amount it must be scrolled in order for you to see its complete contents. The height of the window in pixels is returned in `h` by `calc_dir` and is converted to the number of lines in the window by dividing the height in pixels by the height of a character. You may notice that there is no provision for horizontal slider boxes. This is because we carefully calculate how many columns will fit horizontally in `calc_dir`, thus eliminating the need for horizontal scrolling and making things a little more convenient for the user.

To create the new window, the `new_window` function (discussed below) is called and passed the name array, the slider positions and sizes (`vs`, `hs`, `vp`, and `hp`) that were calculated, and the name and address of a new virtual workstation handle to be used for this window. `new_window` will call the envelope function `setup_window` to create the new window, `wind_set` to make it the current, topmost window, and `clr_display` to clear the background part of the screen from the window. The handle of the new window is returned in `whand`.

Before calling `multi` to process input to the window, the events variable is set to `MU_MESAG` and `MU_KEYBD` since mouse clicks and keypresses are the only types of input this window will accept.

In order to have the directory window redrawn when the user presses a key, the global variable `dir_window` is set to `whand`. This will cause the `got_key` subroutine to call `redo_dir`, which redraws directory windows whenever the screen needs redrawing.

Next, we take advantage of the functions developed in the envelope library that create a GEM window in a regular application program. The `multi` function is called recursively, and it takes over. Notice that the program hasn't returned from the first call to `multi`, so that this call occurs within the first call. In the same way that an application program like the `MandelZoom` program opens a window and accepts input, the directory window will be drawn and mouse and keyboard input will cause events to be passed to `multi`, which will call `was_msg` to handle window operation and `got_key` to handle characters.

Window operations will work normally, since none of the functions called by `multi` can tell that this is not the application's original main window. The program is able to do this because we have been very careful to pass the handles for the virtual workstation and the window, and not let the functions use global window handles.

To close a directory window, the user can type any key, or click on the close box in the window border. `got_key` handles the character input by returning 1, meaning exit, when the `dir_window` variable is nonzero. This causes `multi` to return `do_dir_window`, and we close the window and set `dir_window` 0.

Before `do_dir_window` can return, there's one more thing to consider. If `do_dir_window` is running as a desk accessory, then it is possible it got an `AC_CLOSE` message from GEM while processing the `multi` function. `AC_CLOSE` is sent by GEM when GEM has closed all of an application's windows because some other program has taken over the screen. If this happens, then our program must close the main application window as well as the directory window. A clean way to do this is to arrange for the first call to `multi` to get another `AC_CLOSE` message, replacing the one that the second, nested call to `multi` intercepted.

The `close_me` function, which appears at the end of the Program 6-16, `dodirwind.c`, calls the GEM `appl_write` routine to send the `AC_CLOSE` message to the Shell program's own application id (`gl_apid`). Thus, the next time `evnt_multi` is called, which occurs when the first instance of `multi` loops after `got_key` returns, the `AC_CLOSE` message is received by `evnt_multi` and the application's windows are closed properly.

Program 6-16. dodirwnd.c

```

#include <gemdefs.h>
#include <osbind.h>

#define BYE_BYE -1
#define OBLIVION -2

do_dir_window(count,args,old_wh,old_vw)
int count;
char *args;
int old_wh, old_vw;{

    int whand, vp, hp, vs, hs, x, y, w, h, drv, vw, events, nlines;
    int save_x, save_y, save_w, save_h, wlines, retval;
    static char curdir[100], name[100];
    extern int gl_wchar, gl_hchar, menu_id, dir_window;
    extern char errmsg[];

    drv = Dgetdrv();
    Dgetpath(curdir,drv+1);
    sprintf(name,"%c:%s\\%s", drv+'A', curdir, args );
    sprintf(errmsg,"%d file%s in %s",count,count == 1 ? "" : "s",name);

    wind_get(old_wh,WF_CURRXYWH,&save_x,&save_y,&save_w,&save_h);
    wind_get(0,WF_WORKXYWH,&x,&y,&w,&h);

    vw = old_vw;
    nlines = calc_dir(count, &x, &y, &w, &h);
    if( nlines <= 0 )
        return;

    wlines = h / gl_hchar;
    slide_size( wlines, nlines, &vs );
    slide_size( 1, 1, &hs );
    slide_pos( wlines, nlines, 0, &vp );
    slide_pos( wlines, nlines, 0, &hp );
    whand = new_window(name,1000-vp,hp,vs,hs,x,y,w,h,&vw);
    events = MU_MESAG : MU_KEYBD;
    dir_window = whand;
    retval = multi(events,&whand,0,name,&vw);
    close_window(whand);
    dir_window = 0;
    /*
    ** If the previous call to multi got an AC_CLOSE,
    ** then it returned OBLIVION. We must send another
    ** AC_CLOSE to the multi that is called by main(),
    ** so that the virtual workstation gets handled
    ** properly, and the other window gets closed properly.
    */
    if( retval == OBLIVION )
        close_me();
}

/*
** This routine sends a message to multi, faking an AC_CLOSE.
** This allows routines to be decoupled from actions that
** take place in was_msg(): the caller only needs to know that
** he wants to do whatever action AC_CLOSE causes, without having
** to know anything about the internal workings of was_msg.
*/
close_me(){

    int m[8];
    extern int gl_apid, menu_id, i_am_accessory;

    if( i_am_accessory ){
        m[0] = AC_CLOSE;
    }
}

```

```

m[3] = menu_id;
m[1] = m[2] = m[4] = m[5] = m[6] = m[7] = 0;
appl_write(gl_apid, 16, m);
}

```

The calc_dir function. The purpose of the calc_dir function, Program 6-17, is to put the largest number of filenames from a list into the smallest window that will hold them. For large directories, the window will be the entire desktop work area and the user can move the slider boxes to see the parts that extend past the window borders.

The calc_dir function is called by do_dir_window, which passes it the number of filenames in count, and the dimensions of the desktop area in x, y, w, and h.

calc_dir sets the rows variable to the maximum number of lines possible by dividing the height of the window by the height of one character. Then columns is set to the maximum number of columns possible in the window. A column is 13 characters wide: 8 for the filename, 1 for the period, 3 for the extension, and 1 for a space (the space is replaced by a box containing a diamond shape for a directory name).

If the window is less than 13 characters wide, then columns is 0. To insure that at least part of filename list shows, columns is set equal to one column, and allows the extra characters to be clipped off by the window.

Then calc_dir executes a for loop which searches for the smallest number of columns that will hold the filenames and display as many as it can. The objective is to open the smallest window that displays as many filenames as possible, while keeping the relationship of the window height and width the same, no matter what the window size.

Walking through the loop with sample data will demonstrate how the loop consistently creates a rectangular window with the same proportions, independent of size. If a line is 78 characters long, and the window can hold 10 lines, then rows is 10 and columns will be 6 as the loop is begun. The following table gives the value for each i of the expression:

$i * ((i * \text{rows}) / \text{columns})$

i Value	Resulting Value
0	0
1	1
2	6
3	15
4	24
5	40
6	60
7	77
8	104
9	135
10	160

If there were 30 files to display (count=30), they would be shown in a table with 5 columns, since the loop would stop when *i* reached 5 because the count is less than 40.

To calculate the number of lines in integer arithmetic, count is rounded up by adding columns-1, and dividing by columns. The result is compared to NLINES, which was defined to be the number of lines in the *pl* array, where the list of filenames will be stored. If the number of lines equals or exceeds NLINES, the number is limited to NLINES-1 to prevent the array from overflowing. (You can increase NLINES to accommodate huge directories, but it is already a generous 200 files.)

To determine the final size of the directory window, including the borders, the window dimensions of the area that displays the filenames are put in the variables *tx*, *ty*, *tw*, and *th*, and passed to the GEM *wind_calc* function with the *WC_BORDER* parameter. *wind_calc* calculates the outside dimensions of the window including the borders.

As a programming precaution, we have included some code to confirm that the new window fits on the desktop's work area. The *calc_dir* program is written to be general enough that it's a useful function for other programs. It's possible that the dimensions passed to *calc_dir* are incorrect and would result in an erroneous window size; hence, code is included that double-checks the feasibility of the new window. The available space for a window is obtained with the GEM *wind_get* function. If this space is less than the new window, the window's size is limited to the size returned by *wind_get*.

Finally, *calc_dir* fills in the *pl* array by executing a loop for each line and a loop for each column, calling the *pad* subroutine to add spaces to each filename until it is 13 characters wide. The GEM *strcat* routine copies the data into the array. The array is terminated with a null string when the loops are completed, and *calc_dir* returns the number of lines in the array to *do_dir_window*.

Program 6-17. *calcdir.c*

```
#include <wfparts.h>
#include <gemdefs.h>
#include <document.h>

char pl[NLINES][INCHARS];
int xlines;

calc_dir(count,x,y,w,h)
int count, *x, *y, *w, *h; {

    int i, j, k, columns, rows, tx, ty, tw, th;
    char padded[32];
    extern int gl_wchar, gl_hchar;
    extern char *dir_strs[NFILES];

    rows = *h / gl_hchar;
    columns = *w / gl_wchar / 13;
    if( columns < 1 )
        columns = 1;
    for( i = 0; i < columns; i++ ) {
```

```

        if( count <= i * ((i * rows) / columns) ){
            columns = i;
            break;
        }
    }
    xlines = (count + columns - 1) / columns;
    if( xlines >= NLINES )
        xlines = NLINES - 1;
    tw = columns * 13 * gl_wchar + gl_wchar * 2;
    th = xlines * gl_hchar + gl_hchar;
    tx = (xw - tw) / 2;
    ty = (yh - th) / 2;
    wind_calc(WC_BORDER, WF_PARTS, tx, ty, tw, th, x, y, w, h);
    wind_get( 0, WF_WORKXYWH, &tx, &ty, &tw, &th );
    if( xw > tw ) xw = tw;
    if( xh > th ) xh = th;
    if( x < tx ) x = tx;
    if( y < ty ) y = ty;
    k = 0;
    for( i = 0; i <= xlines; i++ ){
        plfil[0] = ' ';
        plfil[1] = 0;
        for( j = 0; j < columns && k < count; j++, k++ ){
            pad(pad_dir_strs[k], 13);
            strcat(plfil, pad);
        }
        plfil[0] = 0;
        return( xlines );
    }
}

```

The new_window function. The purpose of the new_window function is to create a new window, put it on top of all other windows on the desktop, and make it blank by clearing the background from the window. This function belongs in the envelope library and was included in Chapter 2.

The size of the window to create is passed to new_window from the do_dir_window function, along with the slider box positions and sizes in vp, hp, vs, and hs. The functions you've seen before—setup_window, wind_set, and clr_display—are called to put up the window.

The pad function. The simple pad function takes a string and adds as many characters as specified in the cnt variable that's passed to it. For the directory listings, each filename is padded to 13 characters. It also can be found in Chapter 2 as part of the envelope library.

Printing the Directory with the doit Function

At this point, calc_dir has created an array of filenames in the directory, and the names must now be printed in the window. Let's examine the call graph showing the path of the program through the subroutines.

do_dir_window

new_window Creates a new window

When a new window is created, GEM is notified, and it sends a RE-DRAW message to multi (the second multi call) to handle the input. Continuing with the call graph:

multi	Handles the input (REDRAW)
was_msg	Delegates the action required by input
do_redraw	Handles clipping windows
just_draw	Handles screen refresh and calls got_key with -1
got_key	Handles screen input
redo_dir	Draws directory listing on screen because the global variable <code>dir_window</code> was set by <code>do_dir_window</code> to the new window's handle

This activity results in a window that's the optimum size for the directory listing, and the directory filenames arranged in columns that are scrollable vertically.

The redo_dir function. To copy the array of filenames onto the screen, the redo_dir function is called by got_key. The redo_dir function, Program 6-18, considers the possibility that redo_dir is called more than once, and that the user may have resized the window since do_dir_window created it. As insurance that the window is the right size for the names, redo_dir calls wind_get to get the size of the desktop work area and then calls calc_dir again to recalculate the directory listing. The sliders are set up again, and then print_dir is called to print the filename listing in the window.

Program 6-18. redodir.c

```
# include <gemdefs.h>
# include <document.h>

redo_dir(whand,vw)
int whand, vw;{

    int x, y, w, h, nlines, wlines, wcols, vs, hs;
    extern int dir_index, gl_hchar, gl_wchar;
    int vertical, horizontal, junk;

    hide_mouse();
    just_clear( whand, vw );
    wind_get( whand, WF_WORKXYWH, &x, &y, &w, &h );
    wlines = h / gl_hchar;
    wcols = w / gl_wchar;
    nlines = calc_dir( dir_index, &x, &y, &w, &h );
    slide_size( wlines, nlines, &vs );
    slide_size( wcols, NCHARS, &hs );
    wind_set( whand, WF_VSLSIZE, vs, 0, 0, 0 );
    wind_set( whand, WF_HSLSIZE, hs, 0, 0, 0 );
    print_dir( nlines, whand, vw );
    show_mouse();
}
```

The print_dir function. Using a loop, the print_dir function, Program 6-19, calls the GEM routine v_gtext to copy the array of filenames onto the screen.

Notice that print_dir carefully monitors the envelope's global variables cur_line and cur_col, which are set by the envelope whenever the sliders are changed by the user. This is to insure that the correct portion of the directory is displayed in the window.

Program 6-19. printdir.c

```
# include <gemdefs.h>
# include <document.h>

print_dir(count,whand,vw)
int count, whand, vw;{

    int x, y, w, h, i;
    extern int gl_wchar, gl_hchar, cur_col, cur_line;
    extern char pl[NLINES][NCHARS];

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    just_clear(whand,vw);
    hide_mouse();
    for( i = cur_line; i < count; i++){
        if( strlen( pl[i] ) > cur_col )
            v_gtext(vw, x, y+gl_hchar+(i-cur_line)*gl_hchar,
                    &pl[i][cur_col] );
    }
    show_mouse();
}
```

Building the Desk-Accessory Program

The functions described in this chapter are ready to be linked into an ST desk-accessory program. If you are using a version of C other than *Alcyon C*, included in the *Atari ST Software Developer's Kit*, refer to your *User's Manual* for specific instructions for creating a desk accessory program.

After each of the functions has been compiled, they are linked in the usual way using batch and argument files. The list of filenames for linking is in a file named *linkacc.arg*, and contains the code shown in Program 6-20.

Notice that in the file, *accstart.o* is used instead of *gemstart.o* (used for regular programs), and an extra file, *accsup.o*, is included in the file to define some items that Atari left out of *accstart.o* to save space. Earlier, in Chapter 2, both *accstart.o* and *accsup.o* were listed and discussed.

The batch file to create the library is called *linkacc.bat* and contains the code shown in Program 6-21.

When the program is linked (you'll need to rename the output file to *shell.acc* from *a.prg*), you can boot the desktop from the disk containing this file, and see the Command Shell listed among other desk accessories on the DESK pull-down menu.

Program 6-20. linkacc.arg

```
a.68k=c:accstart.o,main.o,
CALLSYS.O,DOIT.O,BUILTIN.O,CALCDIR.O,CONFIGAC.O,DIRLIST.O,DORM.O,DOMOVE.O,
DODIRWND.O,FINDCMD.O,PRINTDIR.O,PRNTFILE.O,REDODIR.O,GOTKEY.O,DOCOPY.O,
SAVELAST.O,GETHEAD.O,GIVEHELP.O,
accsup.o,env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```

Program 6-21. linkacc.bat

```
c:\bin\link68 [undefined,symbols,command[linkacc.arg]]
c:\bin\relmod a
c:\bin\rm a.68k
c:\bin\wait
```



7 Changing a Desk Accessory to a Regular Program

7 Changing a Desk Accessory to a Regular Program

■ This chapter takes the functions created in Chapter 6 and shows how they can be linked together to form a regular program. Part of the intent of Chapters 6 and 7 is to show how to convert a regular program to a desk accessory, and vice versa. By knowing how to do this conversion, you're not restricted to using a program as one or the other, but can link the object files into an accessory or a program, as your needs dictate.

Turning the Shell functions into a regular program is relatively simple, partly because the functions were written in such a way that no major changes are now needed.

The functions in Chapter 6, and a few additional routines covered here, create a program from which TOS commands can be issued. Mainly, we just need to add some menus and then link the functions using the gemstart.o file instead of the accstart.o and accsup.o routines.

The configap.c File

Accessories begin to differ from regular applications in the configuration file. The configuration information in the configap.c file resembles that in Chapter 6; however, this time the `wind_name` (Command Shell) will be used instead of `access_name`, and the `i_am_accessory` is set to 0, meaning this program is *not* a desk-accessory program type.

Setting the correct name variable and setting the accessory flag to 0 is the extent of the changes required to make this file work for a regular program.

Program 7-1. configap.c

```
#
# include <gemdefs.h>

char *wind_name      = " Command Shell ";

# ifdef USE_RCS
char *resource        = "SHELL.RSC";
# else
```

CHAPTER 7

```
char *resource          = 0;
# endif USE_RCS

char *access_name       = "  Command Shell ";
int i_am_accessory      = 0;
int sx                  = 20; /* small window size */
int sy                  = 50;
int sw                  = 250;
int sh                  = 200;
int slv                 = 0; /* small window vertical slider pos */
int slh                 = 0; /* small window horizontal slider pos */
int svx                 = 1000; /* small window vertical slider size */
int shs                 = 1000; /* small window horizontal slider size */
int min_wide            = 100;
int min_high            = 50;
int interval            = 0;
int events = MU_MESAG : MU_KEYBD;
```

The build_tree Function

Applications that have menus make it convenient for the user to get help, perform file operations, and give users access to the desk accessories.

The menus are built and operated by the `build_tree`, `do_menu`, and `do_main_menu` functions, very similar to the ones used in the MandelZoom and Noise program. The routines presented here are stripped-down versions of those programs that can be inserted in any program that needs menus. For example, they could be used to add menus to the World Map and Plot programs developed earlier in this book.

Using these routines in another program is simply a matter of changing the "Commands" string that appears when the user activates the Help menu item. For example, you might change it to "Map info" in the World Map program. You may also want to add more help topics to the list. And, of course, the `give_help` function that contains the actual help text must be modified to fit the program. The text under the About menu selection in the `do_main_menu` routine also must be changed to reflect the new program.

Program 7-2. bldtree.c

```
# include <gemdefs.h>
# include <obdefs.h>

# define MAXTREE        64
# define M_BLACK        15L /* would be 1, but we changed the color map */

# define TRANSPARENT    0
# define THICK          (long)( 0xFFL << 16 )
# define BOXCOLOR       (long)( M_BLACK << 12 ) | (M_BLACK << 8 )
# define BOXTHIN        (long)( BOXCOLOR | TRANSPARENT | IP_HOLLOW )
# define BOXBITS        (long)( THICK | BOXCOLOR | TRANSPARENT | IP_HOLLOW )
# define LEN            -2 /* Set the width to the length of the string */

# define xx(item) ((t_list[item].ob_x + t_list[item].ob_width) / Wc)
# define yy(item) ((t_list[item].ob_y + t_list[item].ob_height) / Hc)
# define OFFSET        2 /* so the boxes don't about the left edge */

int Wc, Hc;
int About, Quit, Help;
```

```

struct object t_list[MAXTREE];

struct object *
build_tree(){

    extern int gl_wchar, gl_hchar, next_item;
    int root, mbox, desk, file, help;
    int mbox, fbox, obox, hbox, ibox, lbox;
    int lines, desk1, desk2, desk3, desk4, desk5, desk6;

    next_item = 0;
    Hc = gl_hchar;
    Wc = gl_wchar;
    root = addit(t_list, -1, G_IBOX, 0L, 0, 0, 80, 25);

    Hc = gl_hchar + 3;
    lbox = addit(t_list, root, G_BOX, BOXTHIN, 0, 0, 80, 1);

    mbox = addit(t_list, lbox, G_IBOX, 0L, OFFSET, 0, 27, 1);

    desk = addit(t_list, mbox, G_TITLE, " desk ", 0, 0, LEN, 1);
    file = addit(t_list, mbox, G_TITLE, " file ", xx(desk), 0, LEN, 1);
    help = addit(t_list, mbox, G_TITLE, " help ", xx(file), 0, LEN, 1);

    ibox = addit(t_list, root, G_IBOX, 0L, 0, 1, 80, 14);
    Hc = gl_hchar;
    dbox = addit(t_list, ibox, G_BOX, BOXBITS, OFFSET, 0, 19, 8);

    About = addit(t_list, dbox, G_STRING, " Command Shell ", 0, 0, LEN, 1);
    lines = addit(t_list, dbox, G_STRING, "-----", 0, 1, LEN, 1);
    t_list[lines].ob_state = DISABLED;
    desk1 = addit(t_list, dbox, G_STRING, " Desk Accessory 1 ", 0, 2, LEN, 1);
    desk2 = addit(t_list, dbox, G_STRING, " Desk Accessory 2 ", 0, 3, LEN, 1);
    desk3 = addit(t_list, dbox, G_STRING, " Desk Accessory 3 ", 0, 4, LEN, 1);
    desk4 = addit(t_list, dbox, G_STRING, " Desk Accessory 4 ", 0, 5, LEN, 1);
    desk5 = addit(t_list, dbox, G_STRING, " Desk Accessory 5 ", 0, 6, LEN, 1);
    desk6 = addit(t_list, dbox, G_STRING, " Desk Accessory 6 ", 0, 7, LEN, 1);

    fbox = addit(t_list, ibox, G_BOX, BOXBITS, xx(desk)+OFFSET, 0, 6, 1);
    Quit = addit(t_list, fbox, G_STRING, " Quit ", 0, 0, LEN, 1);

    hbox = addit(t_list, ibox, G_BOX, BOXBITS, xx(file)+OFFSET, 0, 11, 1);
    Help = addit(t_list, hbox, G_STRING, " Commands ", 0, 0, LEN, 1);

    if( next_item > 0 )
        t_list[next_item - 1].ob_flags != LASTOB;
    return( t_list );
}

```

The do_menu and do_main_menu Functions

These functions are very basic and can be used with any program to support the menu structure. When the user clicks on a menu item, a message is sent to GEM. GEM sends a message to multi, which calls was_msg, which calls do_menu, which calls do_main_menu. It figures out which command the user selected and then calls the correct function—for example, give_help if a help command was clicked.

The do_menu code is shown in Program 7-3 and should look familiar; likewise, you have seen do_main_menu, Program 7-4, before.

Program 7-3. domenu.c

```
#
# include <obdefs.h>

do_menu(title,item,whand,vw)
int title, item, whand, vw;{

    int ret;
    extern struct object *main_addr;

    ret = do_main_menu(item,whand,vw);
    menu_tnormal(main_addr,title,1);
    menu_tnormal(main_addr,item,1);
    return( ret );
}
```

Program 7-4. domnmenu.c

```
do_main_menu(item,whand,vw)
int item, whand, vw;{

    char str[256];
    extern int About, Quit, Help;

    if( item == About ){
        sprintf(str,"[0][%s!%s!%s!%s!%s][ OK ]",
            " This is a GEM based command ",
            " interpreter. It can execute ",
            " programs, copy, move, print ",
            " or remove files, and it can ",
            " display directories. "
        );
        form_alert(1,str);
        return(0);
    }
    else if( item == Quit ){
        return(1);
    }
    else if( item == Help ){
        give_help(whand,vw);
        return(0);
    }
    sprintf(str,"[0][%s %d][ OK ]","Unknown menu number!",item);
    form_alert(1,str);
    return(0);
}
```

Building the Shell Application

Building the Shell as a regular program is similar to all the other applications we've built. To change the Shell accessory link file to one for a regular program, we use the gemstart.o file instead of accstart.o. We remove accsup.o and we add the filenames bldtree.o, domenu.o, and domnmenu.o to the list.

Program 7-5. link.arg

```
a.68k=c:gemstart.o,main.o,
CALLSYS.o,DOIT.o,BUILTIN.o,CALCDIR.o,CONFIGAP.o,DIRLIST.o,DORM.o,DOMOVE.o,
DODIRWND.o,FINDCMD.o,PRINTDIR.o,PRNTFILE.o,REDODIR.o,GOTKEY.o,DOCOPY.o,
SAVELAST.o,GETHEAD.o,BLDTREE.o,DOMENU.o,DOMNMENU.o,GIVEHELP.o,
env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```



Program 7-6. linkit.bat

```
c:\bin\link68 [undefined,symbols,command[link.arg]]  
c:\bin\reimod a  
c:\bin\rm a.68k  
c:\bin\wait
```

8 Programming the Sound Chip



8 Programming the Sound Chip



8 Programming the Sound Chip

The ST is equipped with a sound chip with which you can create a wide variety of tones including musical notes, drum sounds, and train whistles, similar to the electronic sound capabilities of a synthesizer. In this chapter, you'll see how to program this internal sound chip to generate its full sound assortment.

To explore the sound chip, we'll write a program that puts a two-octave keyboard on the screen and lets a user play it with the top two rows of alphabetic keys on the keyboard. The program also creates an interface that lets a user configure the duration and shape of the notes by moving slider boxes and selecting buttons from a special control panel.

The pull-down menus in the program will include selectable options that let a user change the sound emitted by the sound chip and then display the sound chip's internal registers as numbers. You can use the program as a sound editor and, by knowing the internal register numbers, you can add the sound effects to other programs. Using more menu options, we'll also demonstrate how to create a rhythm section.

As in the other programs in this book, the envelope library is used to interface with functions that were developed in Chapter 2. That library continues to demonstrate its value here by taking care of standard GEM interface programming issues. We have only to provide the functions for this particular application and tailor the standard connecting files and functions, like `config.c` and `got_key`, for our sound chip program.

The `config.c` File

This is, as usual, the first file for this application.

The window name is defined as "Noise!" and the resource and `access_name` variables are set, even though they are not used (just in case you want to make this into a desk accessory). Since this program is not a desk accessory and the Resource Construction Set is not used, `i_am_accessory` and `resource` are set to 0.

This `config.c` file (Program 8-1) is essentially the same as the others in the book except the interval variable is set to 20 milliseconds. This will cause

GEM to send a message to our multi function 50 times each second. You'll see this used when the rhythm section and the clock_tick function are discussed.

Program 8-1. config.c

```
#include <gemdefs.h>

char *wind_name           = " Noise! ";

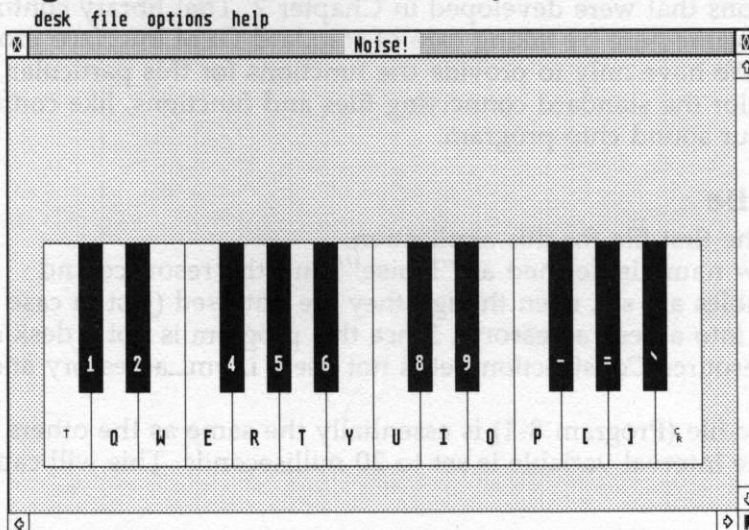
#ifdef USE_RCS
char *resource             = "NOISE.RSC";
#else
char *resource             = 0;
#endif USE_RCS

char *access_name         = " Noise! ";
int i_am_accessory         = 0;
int sx                    = 20;    /* small window size */
int sy                    = 50;
int sw                    = 250;
int sh                    = 125;
int slv                   = 0;     /* small window vertical slider pos */
int slh                   = 0;     /* small window horizontal slider pos */
int sys                   = 1000; /* small window vertical slider size */
int shs                   = 1000; /* small window horizontal slider size */
int min_wide              = 100;
int min_high              = 50;
int interval              = 20;    /* every 50th of a second */
int events = MU_MESAG : MU_BUTTON : MU_KEYBD : MU_M1 : MU_M2;
```

The doit Function

The doit function is responsible for creating the piano keyboard display shown in Figure 8-1. When the user presses a key on the keyboard that corresponds to one of the piano keys, the note is played and the key flickers by rapidly changing to gray, then back to its original color.

Figure 8-1. The Two-Octave Keyboard Created by the doit Function



doit, Program 8-2, calls the `save_screen` function, which is the same as the one used in the MandelZoom program. Note that a default version of `doit` appears in the envelope library and that it must be changed to this default version in order for this program to work properly.

The `doit` routine calls the `show_keys` function to display the piano keyboard. Then it calls `save_screen` to save a copy of this screen so that it can be quickly redrawn when a window or dialog box disappears.

Program 8-2. `doit.c`

```
doit( whand, vw )
int whand, vw;{

    show_keys( whand, vw );
    save_screen(whand);
}
```

The `just_draw` Function

The `just_draw` function used here is exactly the same as the `just_draw` routine used in Chapter 5 with the Mandelbrot set. Therefore, you should copy `justdraw.c` from Chapter 5, Program 5-15, and include it with the other functions in this group.

The `show_keys` Function

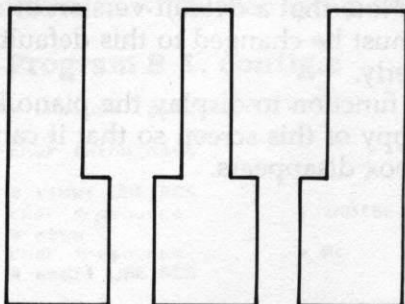
The keyboard and screen background shown in Figure 8-1 are actually drawn by the `show_keys` function listed in Program 8-3.

The first thing `show_keys` does, before drawing the keyboard, is to get the current size of the window with the GEM function `wind_get`. Then it can calculate the keyboard size in relation to the size of the window, enlarging or shrinking the keys accordingly. The size and position of the keys are defined in the variables `white_wide`, `height`, `w_col`, and `b_col`.

Then, `show_keys` calls the `hide_mouse` and `clr_display` functions developed for the envelope (and described earlier in this book) to hide the mouse and clear the screen. With the use of GEM's functions `vsf_color`, `vsf_perimeter`, `vsf_interior`, and `vsf_style`, the background pattern for the keyboard is drawn, setting the color, perimeter, interior fill, and fill style. To fill in the entire window, `fill_box` is called (described below).

With the background set up, the keyboard is next. To draw each key, a loop is executed, calling the `do_white` function to draw the white keys, and the `do_black` function to draw the black keys. To draw the white keys, the three shapes of the white keys must be considered (Figure 8-2).

The position of each type of white key is listed in the array `w_keys`. For each key, the position is passed to the `do_white` function as its fifth parameter. In order to let the user know which key on the keyboard will play which piano key, a letter is passed as the sixth parameter to `do_white`. This information is kept in the array `white_chars`, indexed by the piano key position.

Figure 8-2. The Three Possible Shapes for the White Piano Keys

As with the different shapes of the white keys, special consideration must also be given to the positions of the black keys. Some white keys have no associated black keys; thus the `b_keys` array is used to tell the routine when to draw a black key and when to skip one. The letters needed appear on the black piano keys in the array `black_chars`.

To map the ST keys to the piano keys, and vice versa, two arrays are used. The `askeys` array is indexed by the piano key position and holds the ST key for each piano key. The other array is the `key_info` array with 128 positions, using one for each ASCII character on the keyboard. The structure of the `key_info` array contains three elements: the index for the key, the type (shape), and the pitch. This array is defined in the `keys.h` file. The column on which the left edge of each piano key lies is placed into the `key_info` index element in the loop that draws the keys. Using this index, a piano key can be redrawn when the user presses the ST key mapped to it, providing feedback to the user.

Program 8-3. `showkeys.c`

```
#include <gemdefs.h>
#include <obdefs.h>
#include <keys.h>

int w_keys[] = {
    C,D,E, C,D,D,E, C,D,E, C,D,D,E
};
int b_keys[] = {
    1,1,0,1,1,1,0,1,1,0,1,1,0
};
char black_chars[] = "12 456 89 --";
char white_chars[] = "\tQWERTYUIOP[]\r\177 ";
char askeys[] = "\tQWERTYUIOP[]\r\1771245689--";
int white_wide, height;
struct key_info key_info[128];
int start_y;

show_keys(whand,vw)
int whand, vw;{

    int x, y, w, h, i, w_col, b_col, ch;
    extern gl_wchar, gl_hchar;
```



```

wind_get( whand, WF_WORKXYWH, &x, &y, &w, &h );
start_y = y;
white_wide = w / (NUM_WHITE+1) & ~3;
height = h / 2 & ~3;
w_col = x + (w - NUM_WHITE * white_wide) / 2;
b_col = w_col + white_wide * 3 / 4;
clr_display( whand, vw );
hide_mouse();
vsf_color( vw, RED );
vsf_perimeter( vw, 1 );
vsf_interior( vw, PATTERN );
vsf_style( vw, 1 );
fill_box( vw, x, y, w, h );
for( i = 0; i < NUM_WHITE; i++ ){
    ch = white_chars[i];
    do_white( vw, w_col, white_wide, height, w_keys[i], ch, UP );
    key_info[white_chars[i]].index = w_col;
    if( b_keys[i] ){
        ch = black_chars[i];
        do_black( vw, b_col, white_wide, height, ch, UP );
        key_info[black_chars[i]].index = b_col;
    }
    w_col += white_wide;
    b_col += white_wide;
}
show_mouse();
}

```

The keys.h file. The piano keys have a number of characteristics that must be defined: the shape, the fill pattern, the key's up or down state, and position. The header file listed in Program 8-4 is used to record these characteristics, and then it's included in the functions that need it.

By keeping all the data about the keys in one file, it's easy to change the items when you want to change a key characteristic, such as the color or fill pattern.

This file also defines the structure for the `key_info` array so that it contains three key descriptors: the index to the ST key, the piano key shape, and the note the key's position represents.

In addition, some macros are defined for calculating the dimensions of a key. You'll see these macros used frequently.

Program 8-4. keys.h

```

#
# define NUM_WHITE      14
#
# define HOLLOW          0
# define SOLID           1
# define PATTERN        2
# define HATCH          3
#
# define C              1
# define D              2
# define E              3
#
# define UP             0
# define DOWN          1

struct key_info {

```

```

int index;
int type;
int pitch;
};

# define KLEFT      start
# define KRIGHT     (start+wide)
# define KMIDL      (start+wide/4)
# define KMIDR      (start+wide*3/4)
# define KTOP       (start_y+high-high/6)
# define KMIDY      (KTOP+high*2/3)
# define KBOTTOM    (KTOP+high)
# define KBWIDE     (start+wide/2)

```

The fill_box function. The fill_box function is passed a rectangle, and it converts the x, y, w, and h into a list of points used as arguments to the GEM routine v_fillarea. Although the GEM's v_bar routine—which has a simpler interface (used in the PLOT program)—could have been used, this demonstrates how you can fill more complicated shapes with a more versatile routine.

Note the list of points (x,y pairs) can be longer for different shapes, similar to the v_pline routine in the MAP program in Chapter 3.

Program 8-5. fillbox.c

```

fill_box( vw, x, y, w, h )
int vw, x, y, w, h;{

    int a[32];

    a[0] = x;      a[1] = y;
    a[2] = x + w;  a[3] = y;
    a[4] = x + w;  a[5] = y + h;
    a[6] = x;      a[7] = y + h;
    a[8] = x;      a[9] = y;
    v_fillarea( vw, 5, a );
}

```

The do_white function. To draw the white keys of the piano, the do_white function, Program 8-6, is called and passed the starting column for the key, the width and height, the shape (type) of the key to be drawn, a character to print on the key, and a flag that tells whether the key is up or down.

As part of the setup process, the character that represents the TAB key character on the piano is defined, since the printed TAB character is not commonly recognized. We chose the dagger character because it resembles a T but won't be confused with the letter T that is printed on another key.

Next, the line color for the perimeter line is set to black and the fill parameter is set to 0 to prevent the fill from erasing the perimeter lines. The fill color is white and the interior, solid, if the key is UP. If it's DOWN, the fill color is black and the interior pattern is 4, a light shading. This is how the key is made to flicker when the user presses a key.

In the array a, the points are set to the key shapes (C, D, or E) for which macros are defined in the keys.h file.

Finally, using GEM routines, the key is drawn by calling `v_fillarea` to fill in the key and `v_pline` to outline it, giving the same arguments to both routines. The letter is placed on the key by setting the color to black, the writing mode to transparent (so a white box doesn't appear around the character), and calling `v_gtext` to print the letter. The string `s` that is passed to `v_gtext` contains only the single character that goes on the key. The other arguments to `v_gtext` make sure that the letter is centered on the key.

Before returning, the writing mode is set back to REPLACE with `vswr_mode`.

Program 8-6. dowhite.c

```
#include <keys.h>
#include <obdefs.h>

do_white( vw, start, wide, high, type, ch, up_down )
int vw, start, wide, high, type, ch, up_down;{

    int a[32], x, y, w, h;
    char s[2];
    extern int gl_wchar, gl_hchar, start_y;

    if( ch == '\t' )                /* make tab print nicely */
        ch = ',' ; 0200;          /* daggers look like 't's don't they? */

    s[0] = ch;
    s[1] = 0;
    vsl_color( vw, BLACK );
    vsf_perimeter( vw, 0 );
    if( up_down == UP ){
        vsf_color( vw, WHITE );
        vsf_interior( vw, SOLID );
    }
    else {
        vsf_interior( vw, PATTERN );
        vsf_style( vw, 4 );
        vsf_color( vw, BLACK );
    }
    if( type == C ){
        a[0] = KLEFT;
        a[2] = KMIDR;
        a[4] = KMIDR;
        a[6] = KRIGHT;
        a[8] = KRIGHT;
        a[10] = KLEFT;
        a[12] = KLEFT;
        v_fillarea( vw, 7, a );
        v_pline( vw, 7, a );
    }
    else if( type == D ){
        a[0] = KMIDL;
        a[2] = KMIDR;
        a[4] = KMIDR;
        a[6] = KRIGHT;
        a[8] = KRIGHT;
        a[10] = KLEFT;
        a[12] = KLEFT;
        a[14] = KMIDL;
        a[16] = KMIDL;
        v_fillarea( vw, 9, a );
        v_pline( vw, 9, a );
    }
    else if( type == E ){
        a[0] = KMIDL;
        a[1] = KTOP;
        a[3] = KTOP;
        a[5] = KMIDY;
        a[7] = KMIDY;
        a[9] = KBOTTOM;
        a[11] = KBOTTOM;
        a[13] = KMIDY;
        a[15] = KMIDY;
        a[17] = KTOP;
    }
}
```

```

a[2] = KRIGHT;
a[4] = KRIGHT;
a[6] = KLEFT;
a[8] = KLEFT;
a[10] = KMIDL;
a[12] = KMIDL;
v_fillarea( vw, 7, a );
v_pline( vw, 7, a );
}
vst_color( vw, BLACK );
vswr_mode( vw, MD_TRANS );
v_gtext( vw, KLEFT+wide/2-gl_wchar/2, KBOTTOM-gl_hchar, s );
vswr_mode( vw, MD_REPLACE );
}

```

The do_black function. The do_black function, Program 8-7, operates in essentially the same way as the do_white function, except that it doesn't have to deal with different key shapes.

This time the fill color is set to black and the perimeter to 1, to include the perimeter line in the fill. Also, the character color is set to white so that it will show up on the black key; then the color is reset to black before returning.

Program 8-7. doblack.c

```

#include <keys.h>
#include <obdefs.h>
#include <sliders.h>

do_black( vw, start, wide, high, ch, up_down )
int vw, start, wide, high, ch, up_down;

int a[32];
char s[2];
extern int gl_wchar, gl_hchar, start_y;

s[0] = ch;
s[1] = 0;
vsf_color( vw, BLACK );
vsf_perimeter( vw, 1 );
if( up_down == UP ){
    vsf_interior( vw, SOLID );
}
else {
    vsf_interior( vw, PATTERN );
    vsf_style( vw, 4 );
}

a[0] = KLEFT;
a[2] = KBWIDE;
a[4] = KBWIDE;
a[6] = KLEFT;
a[8] = KLEFT;
v_fillarea( vw, 5, a );
vswr_mode( vw, MD_TRANS );
vst_color( vw, WHITE );
v_gtext( vw, KLEFT+wide/4-gl_wchar/2, KMIDY-gl_hchar, s );
vswr_mode( vw, MD_REPLACE );
vst_color( vw, BLACK );
}

```


The open_data, get_clicks, no_clicks, put_clicks, and do_cleanup Functions

The clicking sound you hear every time you press a key on the ST's keyboard is produced by the sound chip, which receives a signal from the operating system telling it to emit the click. This needs to be disabled so that when the user plays a note on the piano by pressing a key, he or she hears the note instead of the clicking sound.

The key clicks must be turned off early in the program, before a key is pressed. When a key is pressed, the main function in the envelope will call the multi function. main also calls open_data before calling multi, and since it's desirable to disable the key clicks to set up the keyboard, open_data is used to accomplish this task.

The listing shown in Program 8-8 contains the code for open_data and also for the other functions it calls: get_clicks, no_clicks, put_clicks, and do_cleanup.

To turn off the key click, bell, and key-repeat features, three bits need to be cleared. The ST BIOS looks at this byte before generating a noise, and if the bits are cleared it doesn't make any sound. There is a catch that prevents us from just changing the bits: BIOS keeps its data in low memory, where it is protected by the Memory Management Unit, so ordinary programs will bomb if they reference low memory. In order to change the byte that controls the sound chip, the program must be in Supervisor mode.

To solve this problem, the BIOS has a special routine called SUPEXEC (xbios interrupt call number 38) which, when given the name of a function, will set Supervisor mode, call the named function, then reset to User mode.

In open_data SUPEXEC is used to call the get_clicks function, which returns the contents of the protected byte. Then, it uses SUPEXEC to call no_clicks, which clears the three bits and thus turns off the key click, key repeat, and bell.

Once the automatic keyboard sounds have been silenced, the program prepares to draw the piano and play the notes. The open_data function loops through the askeys array, entering the key type (C, D, or E) and pitch into the key_info array. The note pitches are initialized in the pitches array at the start of this file.

Finally, open_data calls the period function (explained below) to set the duration of the note to 10,000, a period we have found produces a pleasant sound that combines the bell and piano tones. This will now be the sound heard when the user strikes a key.

When the program exits, the key clicks, key repeat, and bell are reset. The do_cleanup function uses SUPEXEC to call put_clicks to put back the old value for the key click byte. do_cleanup also prints the bell character, telling the BIOS to ring the bell. The BIOS sets the sound chip back to the default value. This is a convenient way to reset the sound chip as we exit.

Program 8-8. opendata.c

```

#include <keys.h>

int pitches[] = {
/*tab  Q   W   E   R   T   Y   U   I   O   P   [   ]   CR   Del */
/* C   D   E   F   G   A   B   C   D   E   F   G   A   B   C   */
0654,0575,0523,0500,0435,0376,0342,0326,0276,0252,0240,0217,0177,0161,0153,
/* 1   2   3   4   5   6   7   8   9   -   =   '   ,   .   /   */
/* C#  D#   F#  G#  A#   C#  D#   F#  G#  A#   */
0624,0550,    0456,0415,0360,    0312,0264,    0227,0207,0170
};

#define SUPEXEC(x)      (xbios(38,(x)))

char con_info;          /* a place to store key_click ON/OFF info */

open_data(whand,vw,file)
int whand, vw;
char *file;{

    int x, ch;
    extern int get_clicks(), no_clicks(), w_keys[];
    extern char askeys[];
    extern struct key_info key_info[];

    con_info = SUPEXEC(get_clicks);
    SUPEXEC(no_clicks);          /* turn off key clicks */
    for( x = 0; askeys[x]; x++ ){
        ch = askeys[x];
        if( x > NUM_WHITE )
            key_info[ch].type = 0;
        else
            key_info[ch].type = w_keys[x];
        key_info[ch].pitch = pitches[x];
    }
    period(10000);
}

do_cleanup(){

    extern int put_clicks();

    SUPEXEC(put_clicks);
    printf("\n");          /* turn key clicks back on */
                          /* ring bell to reset sound chip */
}

char *conterm = 0x484;

no_clicks(){

    /*
    ** Atari documentation is wrong:
    ** Values for conterm are:
    ** Bit Mask      Function
    ** 0 (01)        Enable key click
    ** 1 (02)        Enable key repeat
    ** 2 (04)        Enable bell
    */
    *conterm &= ~(1+2+4);
}

get_clicks(){

    return(*conterm);
}

put_clicks(){

    extern char con_info;

    *conterm = con_info;
}

```

The got_key Function

Pressing a key causes an event message to be sent to the multi function, which then calls the got_key function, Program 8-9, to handle the key.

If the key is not one of the ST keys superimposed on the piano keyboard, the got_key function returns 0 for "don't exit." If the key is the ESCAPE key (value 033), got_key returns 1 for exit, and the program is exited.

To make sure that the sound registers are configured for the piano, the rest_state function is called. This subroutine, and its complement save_state, keep track of the sound registers that are affected by the rhythm section of the program. Both of these routines will be discussed again later, but for now notice that rest_state is called here to insure that the piano keys don't sound like a snare drum.

The sound chip has three voices, or possible simultaneous sounds it can make. All three voices are needed to have the same tone, or pitch, for each piano note, so each of the three voices is set to the pitch for the key that has been pressed, and play_note is called to start the sound.

To show that a piano key has been played when a corresponding ST key is pressed, got_key calls the do_white or do_black routines with the DOWN parameter to cause the key to be drawn in its shaded, "down" state. By immediately redrawing the key in the UP state, the key is shaded only for an instant, giving the illusion that it has been pressed. Note that when the graphics accelerator chip (the "blitter" chip) is available for the ST, this may happen too fast to see. You may have to add a delay loop between the calls to the do_white or do_black routines with the DOWN parameter to make the "down" state last longer.

The piano keyboard has been arbitrarily ended on the B note because it looks symmetrical that way. However, when you play the two octaves, your ear expects to hear a C as the final note. For this reason, we have made the DEL key make the C note, although there's no piano key corresponding to it.

Program 8-9. gotkey.c

```
# include <keys.h>
```

```
got_key(ch, whand, vw)
int ch, whand, vw;{
```

```
    int type, pitch, col, x, y, w, h;
    extern white_wide, height;
    extern struct key_info key_info[];
```

```
    ch &= 0x7f;                                /* ascii only */
    if( ch >= 'a' && ch <= 'z' )
        ch &= ~040;                            /* convert lower case to upper */
    if( key_info[ch].pitch == 0 )
        return(0);
    if( ch == 033 )
        return(1);
    rest_state();
    tone( 0, key_info[ch].pitch );
    tone( 1, key_info[ch].pitch );
```

```

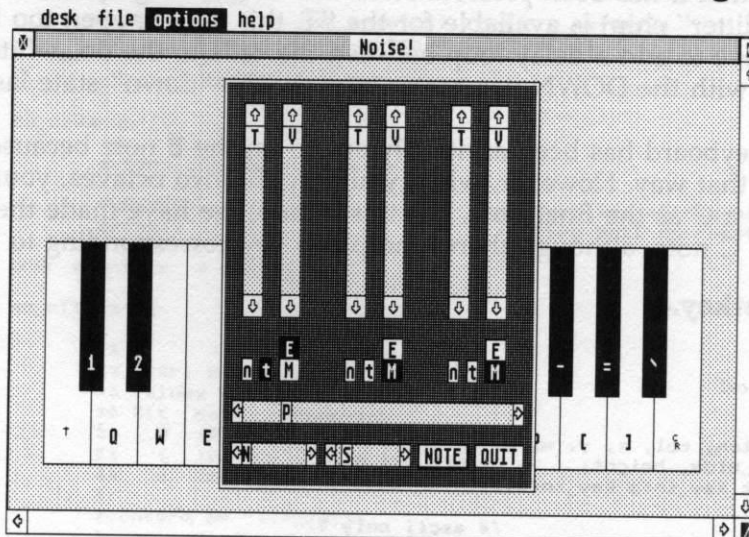
tone( 2, key_info[ch].pitch );
play_note();
if( ch == 0177 )
    return(0); /* keyboard looks better without DEL */
clip_work( whand, vw );
type = key_info[ch].type;
col = key_info[ch].index;
if( type ){
    do_white( vw, col, white_wide, height, type, ch, DOWN );
    do_white( vw, col, white_wide, height, type, ch, UP );
}
else {
    do_black( vw, col, white_wide, height, ch, DOWN );
    do_black( vw, col, white_wide, height, ch, UP );
}
return(0);
}

```

Building an Interactive Sound Control Panel

In addition to the piano keyboard that produces a reasonably good imitation of piano sounds, the application being created here gives the user a method for changing the tones and noises produced by the sound chip. The interface for designing the sounds is a dialog box with sliders and buttons, like the one shown in Figure 8-3.

Figure 8-3. A Sound Control Panel in a Dialog Box



The next functions explained show how to create the menus, sliders, and buttons that produce this interface. By the way, the type of button used is called a *radio button*, which is analogous to the buttons on a car radio. Push one, and another pops out, so only one at a time can be active. The slider

boxes are moved along the slider track by using the mouse to drag the boxes, or clicking on the arrows at the end of each slider.

The captions on the figure describe what each slider and button is for; now let's examine how to draw them and make them work.

The sliders.h Header File

We start by defining a header file that will be included in many of the routines. In the header file we define a structure called `slide` to contain a register value, a minimum and maximum for the slider range, a flag to tell whether a slider is vertical or horizontal, a track and slider index into the tree, and indexes for the arrows at the ends of the slider tracks.

Also in this file are defined macros that name the parts of the dialog box so we can tell which sound register to set. The macro naming patterns such as `S_T_A` and `S_V_B` stand for Slider-Tone-A and Slider-Volume-B.

Program 8-10. sliders.h

```
# define NUM_SLIDERS    12

struct slide {
    int value, min, max, is_vert;
    int track, slider, inc, dec;
};

# define SLIDER                1
# define INC_ARROW             2
# define DEC_ARROW             3
# define NOTE_OP               4
# define QUIT_OP               5

# define S_T_A 0
# define S_T_B 1
# define S_T_C 2
# define S_V_A 3
# define S_V_B 4
# define S_V_C 5
# define S_N   6
# define S_S   7
# define S_P   8
```

The build_tree Function

The object trees for the dialog box and the menu are somewhat complex to set up. The `build_tree` function used in this program is similar to the one in the MandelZoom program, except that it has different menu entries for the file, options and help menus, and it calls another routine, `bld_sliders`, to build a tree for the dialog box that will be used to set the sound chip registers. `build_tree` sets the `next_item` variable to 0 before calling `bld_sliders`, and will set it to 0 again after the call because `build_tree` and `bld_sliders` are adding items to different trees.

During the discussion of `do_main_menu`, we'll talk about the menu selections. The slider boxes inside the dialog box are the topic of interest for now.

Program 8-11. bldtree.c

```

#include <obdefs.h>

#define MAXTREE 64
#define TRANSPARENT 0
#define THICK (long)( 0xFFL << 16 )
#define MBXCOLOR (long)( (BLACK << 12) | (BLACK << 8) )
#define BOXTHIN (long)( MBXCOLOR | TRANSPARENT | IP_HOLLOW )
#define BOXBITS (long)( THICK | MBXCOLOR | TRANSPARENT | IP_HOLLOW )
#define LEN -2 /* Set the width to the length of the string */

#define xx(item) ((t_list[item].ob_x + t_list[item].ob_width) / Wc)
#define yy(item) ((t_list[item].ob_y + t_list[item].ob_height) / Hc)
#define OFFSET 2 /* so the boxes don't abut the left edge */

int About, Quit, Set_snd, Rhythms, Effects, Print_vals;
int HSet_snd, HRhythms, HEffects, HPrint_vals;
int Wc, Hc;

build_tree(){
    extern int gl_wchar, gl_hchar;
    extern int next_item;
    extern struct object *slid_tree, *bld_sliders();
    int root, mbox, desk, file, options, help;
    int dbox, fbox, obox, hbox, ibox, lbox;
    int lines, desk1, desk2, desk3, desk4, desk5, desk6;

    static struct object t_list[MAXTREE];

    next_item = 0;
    Wc = gl_wchar;
    Hc = gl_hchar;
    slid_tree = bld_sliders();

    next_item = 0;
    root = addit(t_list, -1, G_IBOX, 0L, 0, 0, 80, 25);
    Hc += 3;
    lbox = addit(t_list, root, G_BOX, BOXTHIN, 0, 0, 80, 1);
    mbox = addit(t_list, lbox, G_IBOX, 0L, OFFSET, 0, 27, 1);
    desk = addit(t_list, mbox, G_TITLE, " desk ", 0, 0, 0, LEN, 1);
    file = addit(t_list, mbox, G_TITLE, " file ", xx(desk), 0, LEN, 1);
    options = addit(t_list, mbox, G_TITLE, " options ", xx(file), 0, LEN, 1);
    help = addit(t_list, mbox, G_TITLE, " help ", xx(options), 0, LEN, 1);
    ibox = addit(t_list, root, G_IBOX, 0L, 0, 1, 80, 14);
    Hc = gl_hchar;
    dbox = addit(t_list, ibox, G_BOX, BOXBITS, OFFSET, 0, 19, 8);

    About = addit(t_list, dbox, G_STRING, " About Noise ", 0, 0, LEN, 1);
    lines = addit(t_list, dbox, G_STRING, "-----", 0, 1, LEN, 1);
    t_list[lines].ob_state = DISABLED;
    desk1 = addit(t_list, dbox, G_STRING, " Desk Accessory 1 ", 0, 2, LEN, 1);
    desk2 = addit(t_list, dbox, G_STRING, " Desk Accessory 2 ", 0, 3, LEN, 1);
    desk3 = addit(t_list, dbox, G_STRING, " Desk Accessory 3 ", 0, 4, LEN, 1);
    desk4 = addit(t_list, dbox, G_STRING, " Desk Accessory 4 ", 0, 5, LEN, 1);
    desk5 = addit(t_list, dbox, G_STRING, " Desk Accessory 5 ", 0, 6, LEN, 1);
    desk6 = addit(t_list, dbox, G_STRING, " Desk Accessory 6 ", 0, 7, LEN, 1);

    fbox = addit(t_list, ibox, G_BOX, BOXBITS, xx(desk)+OFFSET, 0, 6, 1);
    Quit = addit(t_list, fbox, G_STRING, " Quit ", 0, 0, LEN, 1);

    obox = addit(t_list, ibox, G_BOX, BOXBITS, xx(file)+OFFSET, 0, 16, 4);
    Set_snd = addit(t_list, obox, G_STRING, " Set Sound ", 0, 0, LEN, 1);
    Print_vals = addit(t_list, obox, G_STRING, " Print Values ", 0, 1, LEN, 1);
    Rhythms = addit(t_list, obox, G_STRING, " Rhythms ", 0, 2, LEN, 1);

```

```

Effects = addit(t_list,obox,G_STRING," Sound Effects ",0,3,LEN,1);

hbox = addit(t_list,ibox,G_BOX,BOXBITS,xx(options)+OFFSET,0,16,4);
HSet_snd = addit(t_list,hbox,G_STRING," Set Sound ",0,0,LEN,1);
HPrint_vals = addit(t_list,hbox,G_STRING," Print Values ",0,1,LEN,1);
HRhythms = addit(t_list,hbox,G_STRING," Rhythms ",0,2,LEN,1);
HEffects = addit(t_list,hbox,G_STRING," Sound Effects ",0,3,LEN,1);

if( next_item > 0 )
    t_list[next_item - 1].ob_flags |= LASTOB;
return( t_list );
}

```

The bld_sliders functions. The bld_sliders function, Program 8-12, assembles the dialog box that contains the sliders and buttons shown in Figure 8-3.

To add items into the dialog box object tree s_tree, bld_sliders calls the addit function, which is the same addit routine introduced for the MandelZoom program in Chapter 5. If you're going to enter and run this sound chip program, then be sure to link the addit function into the envelope library.

The first item in the dialog box tree is the box that will hold the buttons and sliders. To make the box be outlined and shadowed, we change bits in its ob_state field after addit has returned.

Next, the NOTE and QUIT buttons are added to the object tree. Their ob_flags bits are set to TOUCHEXIT so the GEM form_do routine will return when the button is clicked. The DEFAULT bit is set on the NOTE button so that a note sounds when the user presses the RETURN key while the dialog box is open.

We want to add sliders for the tone, volume, noise, shape, and period of the three sound chip voices. Sliders are added to the tree by calling the add_slider function, which is explained below. The slider location is passed in the TONEA parameter, defined at the beginning of the file, and we also hand a character for the slider box to add_slider.

The six buttons, one noise button and one tone button for each voice, are added next to enable noise and tone output for each voice. Again, a character is passed for the button and the button location. The button locations are defined at the start of the file, in the variables TONEA, TONEB, and TONEC.

The three radio buttons control the envelope generator on the sound chip. Radio buttons are constructed as buttons within a single parent box. If the RBUTTON flag bit is set, the GEM form_do routine insures that only one button at a time is selected in the box.

The final statements of the bld_sliders function give each slider its proper initial value by calling the all_sliders function. The LASTOB bit in the flags field of the last item in the tree is set to terminate the list and the max_items variable is set to the item number of the last item. Finally, the address of the tree is returned to bld_tree.

Program 8-12. bldslide.c

```

#include <obdefs.h>
#include <sliders.h>

#define BOXFILL ((IP_6PATT << 4) | GREEN)
#define BOXCOLOR ((long)((GREEN << 12) | (GREEN << 8) | BOXFILL))
#define BX 0
#define BY 0
#define BW 33
#define BH 19
#define S_HOR 0
#define S_VER 1

#define N_W 5
#define N_X 21

#define Q_W N_W
#define Q_X (N_X+N_W+1)

#define TOP 2
#define V_WIDE 2
#define H_HIGH 1
#define T_HIGH 8
#define TONE_MIN 0
#define TONE_MAX 1023 /* ( x 4) would be 4095 with LONG sliders */
#define VOL_MIN 0
#define VOL_MAX 15
#define NSE_MIN 0
#define NSE_MAX 31
#define E_P_MIN 0
#define E_P_MAX 4095 /* ( x 16) would be 65535 with LONG sliders */
#define E_S_MIN 8
#define E_S_MAX 15

#define CHAN_WIDE 11
#define TONEA 2
#define VOL_A 6
#define TONEB (CHAN_WIDE+TONEA)
#define VOL_B (CHAN_WIDE+VOL_A)
#define TONEC (CHAN_WIDE*2+TONEA)
#define VOL_C (CHAN_WIDE*2+VOL_A)
#define TGLS_Y 13
#define E_P_Y 15
#define E_P_W 29
#define NSE_Y 17
#define NSE_W 7
#define E_S_X 12
#define E_S_W 7
#define RAX VOL_A
#define RBX VOL_B
#define RCX VOL_C
#define RAY 12
#define GET -1

int note_op, quit_op;

int n_en_a, n_en_b, n_en_c;
int t_en_a, t_en_b, t_en_c;
int r_e_a, r_e_b, r_e_c;
int r_m_a, r_m_b, r_m_c;
int radio_a, radio_b, radio_c;

struct object *
bld_sliders() {
    int box, x;
    static struct object s_tree[ 128 ];

```



```

extern struct slide s_vals[];
extern int next_item, max_item;

next_item = 0;
box = addit(s_tree,-1,G_BOX,BOXCOLOR,BX,BY,BW,BH);
s_tree[box].ob_state != OUTLINED | SHADOWED;
note_op = addit(s_tree,box,G_BUTTON,"NOTE",N_X,NSE_Y,N_W,H_HIGH);
quit_op = addit(s_tree,box,G_BUTTON,"QUIT",Q_X,NSE_Y,Q_W,H_HIGH);
s_tree[note_op].ob_flags = TOUCHEXIT | DEFAULT;
s_tree[quit_op].ob_flags = TOUCHEXIT;

add_slider(s_tree,s_vals,box,S_T_A,TONEA, TOP,V_WIDE,T_HIGH,S_VER,"T");
add_slider(s_tree,s_vals,box,S_T_B,TONEB, TOP,V_WIDE,T_HIGH,S_VER,"T");
add_slider(s_tree,s_vals,box,S_T_C,TONEC, TOP,V_WIDE,T_HIGH,S_VER,"T");

add_slider(s_tree,s_vals,box,S_V_A,VOL_A, TOP,V_WIDE,T_HIGH,S_VER,"V");
add_slider(s_tree,s_vals,box,S_V_B,VOL_B, TOP,V_WIDE,T_HIGH,S_VER,"V");
add_slider(s_tree,s_vals,box,S_V_C,VOL_C, TOP,V_WIDE,T_HIGH,S_VER,"V");

add_slider(s_tree,s_vals,box,S_N,TONEA,NSE_Y,NSE_W,H_HIGH,S_HOR,"N");
add_slider(s_tree,s_vals,box,S_S,E_S_X,NSE_Y,E_S_W,H_HIGH,S_HOR,"S");
add_slider(s_tree,s_vals,box,S_P,TONEA,E_P_Y,E_P_W,H_HIGH,S_HOR,"P");

n_en_a = addit(s_tree,box,G_BUTTON,"n",TONEA,TGLS_Y,1,1);
n_en_b = addit(s_tree,box,G_BUTTON,"n",TONEB,TGLS_Y,1,1);
n_en_c = addit(s_tree,box,G_BUTTON,"n",TONEC,TGLS_Y,1,1);
s_tree[n_en_a].ob_flags != TOUCHEXIT | SELECTABLE;
s_tree[n_en_b].ob_flags != TOUCHEXIT | SELECTABLE;
s_tree[n_en_c].ob_flags != TOUCHEXIT | SELECTABLE;

t_en_a = addit(s_tree,box,G_BUTTON,"t",TONEA+2,TGLS_Y,1,1);
t_en_b = addit(s_tree,box,G_BUTTON,"t",TONEB+2,TGLS_Y,1,1);
t_en_c = addit(s_tree,box,G_BUTTON,"t",TONEC+2,TGLS_Y,1,1);
s_tree[t_en_a].ob_flags != TOUCHEXIT | SELECTABLE;
s_tree[t_en_b].ob_flags != TOUCHEXIT | SELECTABLE;
s_tree[t_en_c].ob_flags != TOUCHEXIT | SELECTABLE;

radio_a = addit(s_tree,box,G_BOX,BOXCOLOR,RAX,RAY,2,2);
radio_b = addit(s_tree,box,G_BOX,BOXCOLOR,RBX,RAY,2,2);
radio_c = addit(s_tree,box,G_BOX,BOXCOLOR,RCX,RAY,2,2);

r_e_a = addit(s_tree,radio_a,G_BUTTON,"E",0,0,2,1);
r_m_a = addit(s_tree,radio_a,G_BUTTON,"M",0,1,2,1);
s_tree[r_e_a].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;
s_tree[r_m_a].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;

r_e_b = addit(s_tree,radio_b,G_BUTTON,"E",0,0,2,1);
r_m_b = addit(s_tree,radio_b,G_BUTTON,"M",0,1,2,1);
s_tree[r_e_b].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;
s_tree[r_m_b].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;

r_e_c = addit(s_tree,radio_c,G_BUTTON,"E",0,0,2,1);
r_m_c = addit(s_tree,radio_c,G_BUTTON,"M",0,1,2,1);
s_tree[r_e_c].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;
s_tree[r_m_c].ob_flags != TOUCHEXIT | SELECTABLE | RBUTTON;

all_sliders(s_tree);

s_tree[next_item-1].ob_flags != LASTOB;
max_items = next_item;
return( s_tree );
}

```

The add_slider function. The `bld_sliders` function calls `add_slider` (Program 8-13) to add each slider to the dialog box tree. A slider consists of four parts: the slider box, the track in which the box moves, and the two arrows in boxes at either end of the track. When `bld_sliders` calls `add_slider`, it passes the dialog box tree, the slider location and orientation (vertical or horizontal), a character to put in the slider button, and pointers to this slider's particulars (`s_vals` and `val_index`) in the array of slider information.

First, the `add_slider` routine adds the track by calling `addit` with parameters that specify a box drawn in `TRKCOLOR`, which is set to black. Next, the arrow boxes are added by giving the object type `G_BUTTON` and the slider box is added by passing `addit` the object type `G_BOXCHAR`. The `G_BOXCHAR` type is a box with a character in it. The character is put into the `ob_spec` field in the object structure, along with the color of the box and character.

The track orientation is controlled by the `is_ver` variable, and determines where on the track the slider box starts and how wide it is. Vertical sliders are two characters wide and horizontal sliders are one character wide, as specified in the parameter to `addit` for the width.

Next, the `ob_flags` bits for the slider boxes and buttons are set to `TOUCHEXIT` so `form_do` will return if the user clicks on them.

The last thing that's needed is to save the tree indexes of the slider's various parts in the appropriate fields of the `s_vals` structure. This lets us cross-reference between the slider values in `s_vals` and the slider appearance as it is defined in the dialog tree. This allows for both reading the value from the sliders when the user changes the settings, and setting the value from the `s_vals` array when we want the program to control the tone and volume.

Program 8-13. addslide.c

```
# include <obdefs.h>
# include <sliders.h>

# define SLDCOLOR      ((long)((0xFFL<<16) | (RED<<12) | (RED<<8)))
# define TRKCOLOR      ((long)((0xFFL<<16) | (BLACK<<12) | (BLACK<<8) | TRKFILL))
# define TRKFILL       ((IP_IPATT << 4) | BLACK)
# define INC_VER        "\1"
# define DEC_VER        "\2"
# define DEC_HOR        "\3"
# define INC_HOR        "\4"

add_slider(s_tree,s_vals,box,val_index,x,y,w,h,is_ver,ch)
struct object *s_tree;
struct slide *s_vals;
int box, val_index, x, y, w, h, is_ver;
char *ch;{

    int track, slider, inc_button, dec_button;
    long int c;

    c = *ch;
    c = SLDCOLOR | (c << 24);
    track = addit(s_tree,box,G_BOX,TRKCOLOR,x,y,w,h);
    if( ! is_ver ){
        inc_button = addit(s_tree,box,G_BUTTON,INC_VER,x,y-1,2,1);
        dec_button = addit(s_tree,box,G_BUTTON,DEC_VER,x,y+h,2,1);
```

```

        slider = addit(s_tree, track, G_BOXCHAR, c, 0, 0, 2, 1);
    }
    else {
        inc_button = addit(s_tree, box, G_BUTTON, INC_HOR, x-1, y, 1, 1);
        dec_button = addit(s_tree, box, G_BUTTON, DEC_HOR, x+w, y, 1, 1);
        slider = addit(s_tree, track, G_BOXCHAR, c, 0, 0, 1, 1);
    }

    s_tree[slider].ob_flags = TOUCHEXIT;
    s_tree[inc_button].ob_flags = TOUCHEXIT;
    s_tree[dec_button].ob_flags = TOUCHEXIT;
    s_vals[val_index].track = track;
    s_vals[val_index].slider = slider;
    s_vals[val_index].inc = inc_button;
    s_vals[val_index].dec = dec_button;
    s_vals[val_index].is_vert = is_ver;
}

```

The all_sliders Subroutine: Getting the Sound Register Values

When `bld_sliders` has finished building the dialog tree box, it calls the `all_sliders` function, Program 8-14. The purpose of this subroutine is to read the sound chip registers and set the slider locations to reflect the register contents.

`bld_sliders` reads the sound chip registers and sets the slider locations by calling the `set_slider` function.

The sound chip has three voices, and therefore has three tone registers. Among the parameters for `set_slider` is a call to the tone routine. To reference the first tone register, `tone` is passed a 0 as well as `GET` to read the value.

The tone register can hold a value from 0 to 4095, which is too large to use conveniently in the slider. This range is divided by shifting the value right by two bits, effectively dividing by 4 and giving a range of 0 to 1024. Although the slider boxes can now only set or display the tone registers in increments of four, 1024 different tones are plenty for this program.

Among the arguments to the `set_slider` function are the address of a slider within the `s_vals` array, the dialog box tree, and a range. The range is defined as `TONE_MIN` and `TONE_MAX`, and the addresses of the different sliders' `s_vals` structures are defined in `sliders.h` as `S_T_A` for SLIDER TONE voice A, `S_T_B` for SLIDER TONE voice B, `S_V_A` for SLIDER VOLUME voice A, and so on.

The same is done for all the sound registers, except for the shape register, which requires some special consideration. Because of the way the hardware is designed, the shape register has 10 meaningful values out of 16 possible values, and 2 of the 10 are redundant.

To simplify things, the redundant choices are eliminated to create a range of 8 to 15. All values less than 4 are changed into 9 and the values 5 through 7 into 15. Now each shape is unique.

Once the shape register values are determined, the shape sliders are built to reflect the changes in the register.

To shade the buttons, indicating that the button is ON (or SELECTED), the `select_on` function is called. Last, the settings of the sound registers are

CHAPTER 8

saved by the `save_state` subroutine, keeping them readily available for `got_key` to insure that the settings are correct for the piano tones.

Program 8-14. `allslide.c`

```
# include <obdefs.h>
# include <sliders.h>

# define GET -1
# define TONE_MIN 0
# define TONE_MAX 1023 /* ( x 4 ) would be 4095 with LONG sliders */
# define VOL_MIN 0
# define VOL_MAX 15
# define NSE_MIN 0
# define NSE_MAX 31
# define E_P_MIN 0
# define E_P_MAX 4095 /* ( x 16 ) would be 65535 with LONG sliders */
# define E_S_MIN 8
# define E_S_MAX 15

all_sliders(s_tree)
struct object *s_tree;{

    unsigned int period();
    int x;
    extern struct slide s_vals[];
    extern int n_en_a, n_en_b, n_en_c;
    extern int t_en_a, t_en_b, t_en_c;
    extern int r_e_a, r_e_b, r_e_c;
    extern int r_m_a, r_m_b, r_m_c;
    extern int radio_a, radio_b, radio_c;

    set_slider(&s_vals[S_T_A],s_tree,tone(0,GET) >> 2,TONE_MIN,TONE_MAX);
    set_slider(&s_vals[S_T_B],s_tree,tone(1,GET) >> 2,TONE_MIN,TONE_MAX);
    set_slider(&s_vals[S_T_C],s_tree,tone(2,GET) >> 2,TONE_MIN,TONE_MAX);
    set_slider(&s_vals[S_V_A],s_tree,volume(0,GET),VOL_MIN,VOL_MAX);
    set_slider(&s_vals[S_V_B],s_tree,volume(1,GET),VOL_MIN,VOL_MAX);
    set_slider(&s_vals[S_V_C],s_tree,volume(2,GET),VOL_MIN,VOL_MAX);
    set_slider(&s_vals[S_N],s_tree,noise(GET),NSE_MIN,NSE_MAX);
    x = shape(GET);
    if ( x < 4 ) /* redundant shapes 0-3 --> 9 */
        x = 9;
    if ( x < 8 ) /* redundant shapes 4-7 --> 15 */
        x = 15;
    shape(x);
    set_slider(&s_vals[S_S],s_tree,shape(GET),E_S_MIN,E_S_MAX);
    set_slider(&s_vals[S_P],s_tree,period(GET) >> 4,E_P_MIN,E_P_MAX);
    select_on( s_tree, noise_enable(0,GET), n_en_a );
    select_on( s_tree, noise_enable(1,GET), n_en_b );
    select_on( s_tree, noise_enable(2,GET), n_en_c );
    select_on( s_tree, tone_enable(0,GET), t_en_a );
    select_on( s_tree, tone_enable(1,GET), t_en_b );
    select_on( s_tree, tone_enable(2,GET), t_en_c );
    select_on( s_tree, mode_bit(0,GET), r_e_a );
    select_on( s_tree, mode_bit(1,GET), r_e_b );
    select_on( s_tree, mode_bit(2,GET), r_e_c );
    select_on( s_tree, mode_bit(0,GET), r_m_a );
    select_on( s_tree, mode_bit(1,GET), r_m_b );
    select_on( s_tree, mode_bit(2,GET), r_m_c );
    save_state();
}
```


The set_slider function. The exact placement of the slider box within the track is done by calling the set_slider function, Program 8-15, which takes a sound register value and converts it to a position for the top or left side of the slider box.

The sound register value and the minimum and maximum values are first stored in the slider value array using the pointer s. Then we get a pointer into the dialog box tree for this slider's object, which happens to be BOXCHAR (a character with a box around it). This is the slider box that slides on the track.

The slider-box position on the track is determined by the sound register value, which was passed to set_slider. The values are scaled to fit the range (max-min) that was established for the height or width of the slider track. The result goes into the ob_y field for a vertical slider, or the ob_x field for a horizontal slider. This x or y value will determine where the left side or top of the slider box will be positioned in the track.

Program 8-15. setslide.c

```
# include <obdefs.h>
# include <sliders.h>

set_slider(s,s_tree,value,min,max)
struct slide *s;
struct object *s_tree;
int value, min, max;{

    struct object *t;
    long int high, wide;
    extern int gl_hchar, gl_wchar;

    s->value = value;
    s->min = min;
    s->max = max;
    t = &s_tree[s->slider];
    high = value - min;
    high *= s_tree[s->track].ob_height - gl_hchar;
    high /= max - min;
    wide = value - min;
    wide *= s_tree[s->track].ob_width - gl_wchar;
    wide /= max - min;
    if( s->is_vert )
        t->ob_y = high;
    else
        t->ob_x = wide;
}
```

The do_menu Function

When the user selects a menu item, the multi function calls the do_menu function, Program 8-16.

You may recognize this function as the same as the do_menu version in the MandelZoom program. It's important that this version be used instead of the default version in the envelope library. From here, the program calls the do_main_menu function to handle the menu item the user has selected.

Program 8-16. domenu.c

```
#
# include <obdefs.h>

do_menu(title,item,whand,vw)
int title, item, whand, vw;{

    int ret;
    extern struct object #main_addr;

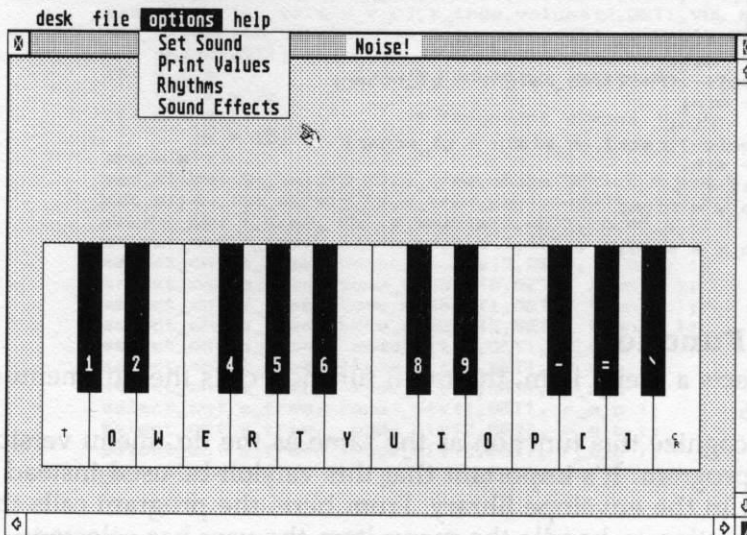
    ret = do_main_menu(item,whand,vw);
    menu_tnormal(main_addr,title,1);
    menu_tnormal(main_addr,item,1);
    return( ret );
}
```

The do_main_menu Function

The do_main_menu function (Program 8-17) is notified that a menu item has been selected, and then determines what action to take depending on the menu.

Selecting the menu items About or one of the Help menu items causes do_main_menu to call the GEM form_alert routine to display the information that we provide. If the user selects the Quit menu item, then the value 1 is returned, causing the multi function to exit.

The other menu items that went into the menu tree built earlier with bld_tree produce the menu shown in Figure 8-4, and are the other four possible menu choices for the user.

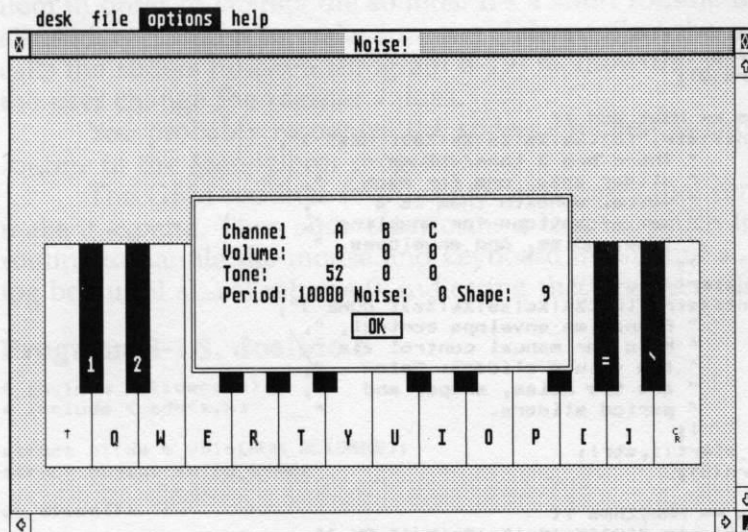
Figure 8-4. The Four Options

When Select sound (Set_snd) is selected, the do_slide function is called to put up the dialog box of sliders and buttons that let the user change the sound chip registers. The registers are saved in memory with the save_state routine so they can be restored should they be changed temporarily—for example, when the rhythm option uses the registers to emit drum sounds.

By selecting Rhythms, the program calls the do_rhythms function to produce rhythmic beat patterns. do_rhythms is discussed later, and you'll see that each time do_rhythms is called, it changes the current rhythm being played. It starts with silence and cycles through all of the patterns, then starts over again with no rhythm.

The Print Values (Print_vals) menu item prints the register contents in a window, as numbers that you can copy down and use in programs. A programmer can design a sound by moving the slider boxes around, then select this menu item to see the register values. This window showing the register contents appears in Figure 8-5.

Figure 8-5. The Window Showing the Values of the Sound Chip Registers



Selecting Sound Effects (Effects) produces a sampler of noises—primarily gunshots and explosions such as you might want to use in game programs.

Program 8-17. domnmenu.c

```
do_main_menu(item,whand,vw)
int item, whand, vw;

char str[256];
extern int About, Quit, Set_snd, Rhythms, Effects, Print_vals;
```

CHAPTER 8

```

extern int HSet_snd, HRhythms, HEffects, HPrint_vals;

if( item == About ){
    sprintf(str, "[0][%s:%s:%s:%s][ OK ]",
        " Noise! Control the sound ",
        " chip in your Atari ST. ",
        " Set the chip's registers, ",
        " play a piano, ",
        " or percussion rhythms. "
    );
    form_alert(1, str);
    return(0);
}

else if( item == Quit ){
    return(1);
}

else if( item == Set_snd ){
    do_slide();
    save_state();
    return(0);
}

else if( item == Rhythms ){
    do_rhythms();
    return(0);
}

else if( item == Print_vals ){
    print_vals();
    return(0);
}

else if( item == Effects ){
    do_effects();
    return(0);
}

else if( item == HSet_snd ){
    sprintf(str, "[0][%s:%s:%s:%s][ NEXT ]",
        " There are 3 tone/volume ",
        " slider sets, one for each ",
        " voice. Beneath them is a ",
        " set of buttons for enabling ",
        " tone, noise, and envelopes. "
    );
    form_alert(1, str);
    sprintf(str, "[0][%s:%s:%s:%s][ DONE ]",
        " E enables envelope control, ",
        " M is for manual control via ",
        " the volume sliders. Below ",
        " are the noise, shape, and ",
        " period sliders. "
    );
    form_alert(1, str);
    return(0);
}

else if( item == HRhythms ){
    sprintf(str, "[0][%s:%s:%s:%s][ OK ]",
        " Each time you click on ",
        " the rhythm menu item, ",
        " a new rhythm begins. ",
        " Loop through all of ",
        " them, and they stop. "
    );
    form_alert(1, str);
    return(0);
}

else if( item == HEffects ){
    sprintf(str, "[0][%s:%s:%s:%s][ OK ]",
        " Select the effects menu ",
        " item to hear some sound "
    );

```



```

        " effects such as gunshots, ",
        " explosions, and other      ",
        " varieties of mayhem.        "
    );
    form_alert(1,str);
    return(0);
}

else if( item == HPrint_vals ){
    sprintf(str,"[0][%s!%s!%s!%s!%s][ OK ]",
        " After setting up a sound, ",
        " select the print values    ",
        " item to print out the      ",
        " sound register contents.    ",
        "                             "
    );
    form_alert(1,str);
    return(0);
}

sprintf(str,"[0][%s %d][ OK ]","Unknown menu number!",item);
form_alert(1,str);
return(0);
}

```

The do_slider and sliders functions. The do_slider function (Program 8-18) is called by do_main_menu when the user selects the Set Sound menu item in order to change the sounds. It's a short routine that first calls the all_sliders function to make sure the sliders reflect the current sound, and then calls the sliders function (Program 8-19) to show the slider dialog box and let the user change the register values.

You probably recognize the sliders function as similar to the dialog sub-routine in the Mandelbrot program.

The GEM routines form_center and form_dial center the dialog box and make it expand. Then sliders performs a loop in which it calls GEM's form_do routine to handle the mouse and keyboard input, and sl_set to set up the dialog box until sl_set returns 0, indicating that the user clicked the QUIT button.

Program 8-18. doslide.c

```

#include <sliders.h>
#include <obdefs.h>

struct slide s_vals[NUM_SLIDERS];
struct object *slid_tree;

do_slide(){
    all_sliders(slid_tree);
    sliders( slid_tree, 0, s_vals );
}

```

Program 8-19. sliders.c

```

#include <obdefs.h>
#include <gemdefs.h>

sliders(box_tree,field,vals)
struct object *box_tree;
int field;

```

```

struct slide #vals;{

    int x, y, w, h;
    int littlex, littley, littlew, littleh;
    int operation;

    if( field < 0 )                /* Atari doc is wrong */
        field = 0;                /* -1 blows up, should be 0 or valid */
    form_center(box_tree,&x,&y,&w,&h);
    littlew = littleh = 50;
    littlex = x + w / 2 - littlew;
    littley = y + h / 2 - littleh;
    form_dial(FMD_START,littlex,littley,littlew,littleh,x,y,w,h);
    form_dial(FMD_GROW,littlex,littley,littlew,littleh,x,y,w,h);
    objc_draw(box_tree,0,9,x,y,w,h);
    do {
        operation = form_do(box_tree,field);
    } while( sl_set(box_tree,field,vals,operation) );
    form_dial(FMD_SHRINK,littlex,littley,littlew,littleh,x,y,w,h);
    form_dial(FMD_FINISH,littlex,littley,littlew,littleh,x,y,w,h);
    return( operation );
}

```

The sl_set Function

Inside the dialog box, there are five different types of items: the sliders, the toggle buttons, the radio buttons, the QUIT button, and the NOTE button. The `sl_set` function, Program 8-20, organizes the different types and how they are handled.

To determine which object was modified, the `which_one` function is called. If the object was one of the noise- or volume-enable (`n_en_a` and `t_en_c`) toggle buttons, it is taken care of by calling the `toggles` function. The radio buttons are handled with the `rad_button` function. These routines are discussed in more detail later.

For the QUIT button, the function simply returns 0 so the sliders function will break out of its loop.

For the NOTE button, the slider-box positions in their tracks are updated, and the `play_note` is called to make the speaker generate the current sound.

When the user moves the slider boxes or clicks on the arrows, the `set_slider` subroutine is called to move the slider; then `slid_val`, to update the registers and play the note; and finally the GEM `objc_draw` routine, to draw the slider in its new position.

When the user drags a slider box, the `sl_set` routine calls the GEM `graf_slidebox` routine to handle the slider's movement. It draws a small rectangle that follows the mouse up and down the slider track. When the mouse button is released, letting go of the slider box, `graf_slidebox` returns a value from 0 to 1000, indicating the box's position. This is scaled in the usual way to a value between `s->min` and `s->max`, the minimum and maximum values allowed for a particular slider. This value is then passed to `set_slider`.

When the user clicks on the arrows at either end of a slider track, the function checks to see if there's room to increment or decrement the value in

the range, and then it calls `set_slider` with the new value so it can convert it to the new position for the slider box.

Program 8-20. `slset.c`

```
#include <obdefs.h>
#include <sliders.h>

int max_items      = 0;

sl_set(tree,field,vals,operation)
struct object *tree;
int field;
struct slide *vals;
int operation;{

    struct slide *s;
    struct object *p;
    int val_index, type, val, step;
    long l, range;
    extern int t_en_c, n_en_a;

    p = tree;
    if( operation < 0 || operation > max_items )
        return(1);
    if( operation >= n_en_a && operation <= t_en_c ){
        toggles( operation, tree );
        all_sliders(tree);
        play_note();
        return(1);
    }
    if( tree[operation].ob_flags & RBUTTON ){
        rad_button( operation, tree );
        all_sliders(tree);
        play_note();
        return(1);
    }
    val_index = which_one( vals, operation, &type );
    if( type == NOTE_OP ){
        tree[operation].ob_state &= ~SELECTED;
        all_sliders(tree);
        play_note();
        return(1);
    }
    if( type == QUIT_OP )
        return(0);
    s = &vals[val_index];
    range = s->max - s->min;
    switch( type ){
        case SLIDER:
            l = graf_slidebox( tree,s->track,s->slider,s->is_vert);
            l %= range;
            l += 500;
            l /= 1000;
            val = s->min + l;
            set_slider(s,tree,val,s->min,s->max);
            break;
        case DEC_ARROW:
            if( s->value < s->max )
                set_slider(s,tree,s->value+1,s->min,s->max);
            break;
        case INC_ARROW:
            if( s->value > s->min )
                set_slider(s,tree,s->value-1,s->min,s->max);
            break;
    }
```

```

    }
    slid_val(tree, val_index, s->value);
    objc_draw(p, s->track, 9, p->ob_x, p->ob_y, p->ob_width, p->ob_height);
    return(1);
}

```

The which_one function. The which_one subroutine converts a tree index into a slider index. In this application, it's called by sl_set to determine which object the user has modified.

It checks for the object type NOTE_OP and QUIT_OP, then loops through the array of slider information, looking for a match between the index of the object selected in the dialog tree and the dialog tree indices stored in the slider array. When a match is found, the type variable is filled in and the slider index is returned to the sl_set function.

Program 8-21. whichone.c

```

#include <sliders.h>

/*
** Which_one takes an index into an object tree, and returns
** an index into the value array for the sliders. It puts the
** type of the operation (SLIDER, INC_ARROW, DEC_ARROW) into type.
*/
which_one( vals, operation, type )
struct slide *vals;
int operation, *type;{

    int x;
    extern int note_op, quit_op;

    if( operation == note_op ){
        *type = NOTE_OP;
        return(-1);
    }

    if( operation == quit_op ){
        *type = QUIT_OP;
        return(-1);
    }

    for( x = 0; x < NUM_SLIDERS; x++ ){
        if( vals[x].slider == operation ){
            *type = SLIDER;
            return(x);
        }
        else if( vals[x].inc == operation ){
            *type = INC_ARROW;
            return(x);
        }
        else if( vals[x].dec == operation ){
            *type = DEC_ARROW;
            return(x);
        }
    }

    return(-1);
}

```


The save_state and rest_state Functions

The sound chip register settings need to be saved so they can be restored. When the user plays the rhythm section of this application, the sound chip is set to the rhythm sounds. When the rhythm section is finished, the user should be able to return to the dialog box and find it unchanged. The save_state and rest_state functions (Program 8-22), which are complementary routines, share the same data about the register values. They are only concerned with the registers that are changed by the rhythm section.

When sl_set returns 0 to the sliders function, sliders closes the dialog box and returns to do_slide, which returns to do_main_menu. The do_main_menu routine calls save_state to copy the sound chip's register values into memory.

Program 8-22. savstate.c

```
# define GET    -1

int t1, t2, t3, p, n, te1, te2, te3, ne1, ne2, ne3;

save_state(){
    t1 = tone(0,GET);
    t2 = tone(1,GET);
    t3 = tone(2,GET);
    te1 = tone_enable(0,GET);
    te2 = tone_enable(1,GET);
    te3 = tone_enable(3,GET);
    ne1 = noise_enable(0,GET);
    ne2 = noise_enable(1,GET);
    ne3 = noise_enable(2,GET);
    p = period(GET);
    n = noise(GET);
}

rest_state(){
    tone(0,t1);
    tone(1,t2);
    tone(2,t3);
    tone_enable(0,te1);
    tone_enable(1,te2);
    tone_enable(3,te3);
    noise_enable(0,ne1);
    noise_enable(1,ne2);
    noise_enable(2,ne3);
    period(p);
    noise(n);
}
```

Notes on the Sound Chip Registers

The ST's sound chip has 16 read/write control registers. The registers control the tone, volume, period, and shape of the note and noise that construct the sound emitted from the speaker. Note that the last two registers control I/O (not sound) and, consequently, are of no interest to us here.

The following descriptions of the functions that set these registers will be easier to understand if you refer to Figure 8-15 to see how the registers are used.

The play_note Function

Reading the shape register and then putting the same value back into the register will generate a sound. This is what the play_note function, Program 8-23, does when it's called by sl_set.

The GEM BIOS routine Giaccess is used to read the shape register, which is register 13 (called R15 because the registers are numbered in octal instead of decimal). Then the shape register is set to the same value with the SET_THE_REG argument to Giaccess. Because the argument is other than GET (-1), the register is set instead of read.

Program 8-23. playnote.c

```
# include <osbind.h>
# define GET          -1

# define SET_THE_REG  0x80

# define SHAPEREG     015

play_note(){
    int x;

    x = Giaccess(0,SHAPEREG);
    Giaccess(x,SHAPEREG | SET_THE_REG);
}
```

The toggles and select_on Functions

The toggles function, Program 8-24, takes advantage of the fact that the noise and tone information are arranged sequentially in the object tree array s_tree by the bld_sliders function.

The index into the array tells whether this was a tone or noise button that the user toggled, and by subtracting the selected object's index from the first tone or noise-enable indices into the array, we find out which voice is affected—voice 0, 1, or 2.

Once the voice number is known, tone_enable or noise_enable is called to "toggle" the voice register on or off, depending on its current state. To toggle the register, the opposite of the current value is passed to tone_enable or noise_enable, so that if the register was set to 1, it is reset to 0, and vice versa.

Then the select_on function is called with the current register value to make the button be drawn dark for selected (on), or white for unselected (off). The GEM routine objc_draw then redraws the button in its new state.

The select_on function, Program 8-25, sets or clears the SELECTED bit in the ob_state field of an object, depending on the value passed in the true_false argument.

Program 8-24. toggles.c

```
# include <obdefs.h>

# define GET          -1

toggles( index, p )
int index;
struct object *p;{

    int i;
    extern int t_en_a, n_en_a;

    if( index >= t_en_a ){
        i = index - t_en_a;
        tone_enable( i, !tone_enable(i,GET) );
        select_on( p, tone_enable(i,GET), index );
    }
    else {
        i = index - n_en_a;
        noise_enable( i, !noise_enable(i,GET) );
        select_on( p, noise_enable(i,GET), index );
    }
    objc_draw(p,index,9,p->ob_x,p->ob_y,p->ob_width,p->ob_height);
}
```

Program 8-25. selecton.c

```
# include <obdefs.h>

select_on( s_tree, true_false, index )
struct object *s_tree;
int true_false, index;{

    if( true_false )
        s_tree[index].ob_state != SELECTED;
    else
        s_tree[index].ob_state &= ~SELECTED;
}
```

The tone_enable and noise_enable Functions

The tone_enable and noise_enable functions (Programs 8-26 and 8-27) set or read the tone-enable or noise-enable bits from the sound chip.

When the tone_enable or noise_enable functions are passed the value GET, they return the current value from the sound chip registers. Otherwise, the register is set to the value that was passed. The GEM BIOS routine Giaccess is used to access the registers. If the register number passed to Giaccess has the high bit set by adding SET_THE_REG, then the register is set; otherwise its current value is returned.

Program 8-26. toneenab.c

```
# include <osbind.h>
# define GET          -1
# define SET_THE_REG  0x80
```

CHAPTER 8

```
# define MIXERREG      07

/*
** Read or set the tone enable bit in the mixer register.
** Note that turning a bit in the mixer ON turns the feature OFF.
*/
tone_enable(channel,on_off)
int channel, on_off;{

    int old_val;

    if( on_off == GET )
        return( !((Giaccess(0,MIXERREG) >> channel) & 1));

    old_val = Giaccess(0,MIXERREG);
    if( on_off )
        old_val &= ~(1 << channel);    /* turn the bit OFF */
    else
        old_val |= 1 << channel;        /* turn the bit ON */
    Giaccess( old_val, MIXERREG + SET_THE_REG );
}
```

Program 8-27. noisenab.c

```
# include <osbind.h>
# define GET          -1

# define SET_THE_REG  0x80
# define MIXERREG      07

/*
** Read or set the noise enable bit in the mixer register
** Just like tone_enable, but the noise bits are three bits to the left
** of the tone bits.
** Note that turning a bit in the mixer ON turns the feature OFF.
*/
noise_enable(channel,on_off)
int channel, on_off;{

    int old_val;

    if( on_off == GET )
        return( !((Giaccess(0,MIXERREG) >> (channel+3)) & 1));
    old_val = Giaccess(0,MIXERREG);
    if( on_off )
        old_val &= ~(1 << (channel+3)); /* turn the bit OFF */
    else
        old_val |= 1 << (channel+3);    /* turn the bit ON */
    Giaccess( old_val, MIXERREG + SET_THE_REG );
}
```

The rad_button Function

When a radio button is selected, the `rad_button` function is called. The radio buttons for each voice determine whether the envelope generator will control the volume, or whether the volume will be set manually by the VOLUME slider boxes.

Each voice has its own pair of radio buttons, kept in parent boxes called `radio_a`, `radio_b`, and `radio_c`. When a button is selected, GEM changes the `SELECTED` bit, and the `rad_button` subroutine displays the changed button.

When `rad_button` is passed a button index, it calls `mode_bit` to select a mode bit in the sound chip's MIXER register and change it. It then sets the index to the parent box of the radio buttons affected, and calls the `objc_draw` subroutine to draw the subtree containing the parent and its two children.

Program 8-28. `radbutn.c`

```
#include <obdefs.h>

rad_button( index, p )
int index;
struct object *p;{

    extern int r_e_a, r_e_b, r_e_c, r_m_a, r_m_b, r_m_c;
    extern int radio_a, radio_b, radio_c;

    if( index == r_e_a ){
        mode_bit( 0, 1 );
        index = radio_a;
    }
    else if( index == r_e_b ){
        mode_bit( 1, 1 );
        index = radio_b;
    }
    else if( index == r_e_c ){
        mode_bit( 2, 1 );
        index = radio_c;
    }
    else if( index == r_m_a ){
        mode_bit( 0, 0 );
        index = radio_a;
    }
    else if( index == r_m_b ){
        mode_bit( 1, 0 );
        index = radio_b;
    }
    else if( index == r_m_c ){
        mode_bit( 2, 0 );
        index = radio_c;
    }
    objc_draw(p,index,9,p->ob_x,p->ob_y,p->ob_width,p->ob_height);
}
```

The `slid_val` Function

Moving a slider box calls the `slid_val` function with an index telling which slider changed and its new positional value.

`slid_val` calls the `tone` function to set one of the three tone registers, or the volume subroutine for one of the three volume registers, or the noise, shape, or period subroutines to set those registers. These functions, which all look somewhat similar, are explained in the next sections.

Last, it calls the `all_sliders` routine to update the slider values, and then `play_note` to produce a sound as feedback to the user.

Program 8-29. slideval.c

```

#include <obdefs.h>
#include <sliders.h>

slid_val(tree, val_index, value)
struct object *tree;
int val_index, value; {

    unsigned int period();

    switch( val_index ){
        case S_T_A:
            tone(0, value << 2);
            break;
        case S_T_B:
            tone(1, value << 2);
            break;
        case S_T_C:
            tone(2, value << 2);
            break;
        case S_V_A:
            volume(0, value);
            break;
        case S_V_B:
            volume(1, value);
            break;
        case S_V_C:
            volume(2, value);
            break;
        case S_N:
            noise(value);
            break;
        case S_S:
            shape(value);
            break;
        case S_P:
            period(value << 4);
            break;
    }
    all_sliders(tree);
    play_note();
}

```

The tone function. To vary the tone of the sound register, the tone function, Program 8-30, has to set two registers: one for coarse tone and one for fine.

The fine tone is the low 8 bits and the coarse tone is the high 4 bits of the 12-bit tone value. These are selected by adding two times the voice number (channel) to the TONEREGS constant, which selects which register set to use.

The value GET is used as an argument for the GEM BIOS Giaccess routine, which accesses the registers. If GET equals -1, the current value is returned; if any other values are used, then the fine and coarse tone registers are set.

Program 8-30. tone.c

```
#include <osbind.h>
#define GET -1

#define SET_THE_REG 0x80

/*
** The six tone registers are 0 and 1 for channel A,
** 2 and 3 for channel B, and
** 4 and 5 for channel C. The even register is the fine tune register,
** and the odd register is the coarse tune register.
*/
#define TONEREGS 0

tone(channel,value)
int channel, value;{

    int coarse, fine;

    if( value == GET ){ /* just return the current value */
        coarse = Giaccess(0,TONEREGS+channel*2+1);
        fine = Giaccess(0,TONEREGS+channel*2 );
        value = ((coarse & 0xf) << 8) | fine;
        return( value );
    }

    /*
    ** Set the value
    */
    coarse = (value >> 8) & 0xf;
    fine = value & 0xff;
    Giaccess(coarse,TONEREGS+channel*2+1+SET_THE_REG);
    Giaccess(fine,TONEREGS+channel*2+SET_THE_REG);
}
```

The volume and mode_bit functions. The fifth bit in the amplitude (volume) register is the envelope control bit, which is not affected by changing the volume slider. Thus, the volume subroutine has to isolate the low four bits from the envelope control bit.

To change only the four volume bits, the volume function, Program 8-31, performs an AND operation on the register value with ENV_CONTROL, which has been defined to have a bit set only in bit 5 (hexadecimal 0x10). Then an OR operation is performed on the result and the low four bits of the value, and the register is set to the resulting value.

The envelope generator is controlled by the fifth bit in the volume register, which is set and cleared with the mode_bit function, Program 8-32. Setting and clearing it is simply a matter of using the AND and OR operations on the ENV_CONTROL bit in the register for the proper voice.

Program 8-31. volume.c

```
#include <osbind.h>
#define GET -1

#define SET_THE_REG 0x80

/*
** The three volume registers are 10, 11, and 12, for channels A, B, and C
*/
```

```

#define VOLREGS      010
#define ENV_CONTROL  0x10

volume(channel,value)
int channel, value;{

    int gi_val;

    gi_val = Giaccess(0,VOLREGS+channel);
    if( value == GET ){ /* just return the current value */
        return( gi_val & 0xf );
    }
    value = (gi_val & ENV_CONTROL) | (value & 0xf);
    /*
    ** Set the value
    */
    Giaccess(value,VOLREGS+channel+SET_THE_REG);
}

```

Program 8-32. modebits.c

```

#include <osbind.h>

#define GET          -1
#define VOLREGS      010
#define ENV_CONTROL  0x10
#define SET_THE_REG  0x80
/*
** Read or set the amplitude mode bit in the amplitude register
*/
mode_bit(channel,on_off)
int channel, on_off;{

    int old_val;

    if( on_off == GET )
        return( (Giaccess(0,VOLREGS+channel) & ENV_CONTROL) != 0 );

    old_val = Giaccess(0,VOLREGS+channel);
    if( on_off )
        old_val |= ENV_CONTROL;
    else
        old_val &= ~ENV_CONTROL;
    Giaccess( old_val, VOLREGS + channel + SET_THE_REG );
}

```

The noise, period, and shape functions. The noise function (Program 8-33) sets the 5-bit noise register (R6) in a manner similar to the tone and volume functions.

The period function (Program 8-34) has to set two registers like the tone subroutine—a coarse and fine register. These registers are eight bits each, and are shifted and ORed to set or read the period value.

The shape function (Program 8-35) changes the shape register, causing the sound chip to express itself. The shape register (R15) has 4 bits, but we use only the high 8 values out of the 16 that are possible, since the low 8 are redundant. The shape registers are illustrated back in Figure 8-5, showing which registers create redundant shapes.

Program 8-33. noise.c

```
#include <osbind.h>
#define GET -1

#define SET_THE_REG 0x80
#define NOISEREG 06

/*
** Set or return the value in the noise register
*/
noise(value)
int value;{

    if( value == GET ){
        value = Giaccess(0,NOISEREG) & 0x1f;
        return( value );
    }

    Giaccess( value & 0x1f, NOISEREG + SET_THE_REG );
}
```

Program 8-34. period.c

```
#include <osbind.h>
#define GET -1

#define SET_THE_REG 0x80

/*
** The envelope period registers are 13 and 14, for fine and coarse tune.
*/
#define PERIODREG 013

unsigned int
period(value)
unsigned int value;{

    unsigned int coarse, fine;

    if( value == GET ){ /* just return the current value */
        coarse = Giaccess(0,PERIODREG+1);
        fine = Giaccess(0,PERIODREG);
        value = ((coarse & 0xff) << 8) | fine;
        return( value );
    }

    /*
    ** Set the value
    */
    coarse = (value >> 8) & 0xff;
    fine = value & 0xff;
    Giaccess(coarse,PERIODREG+1+SET_THE_REG);
    Giaccess(fine,PERIODREG+SET_THE_REG);
}
```

Program 8-35. shape.c

```
#include <osbind.h>
#define GET -1

#define SET_THE_REG 0x80
#define SHAPEREG 015

shape(value)
```

```
int value;{
    if( value == GET ){
        value = Giaccess(0,SHAPEREG);
        return( value );
    }
    Giaccess(value & 0xf, SHAPEREG+SET_THE_REG);
}
```

The print_vals Function: The Print Values Menu Selection

When the user selects Print Values from the menu, the `do_main_menu` function calls the `print_vals` function, Program 8-36. `print_vals` calls the `GEM form_alert` function to display a window with the character strings in the `str` array, and the register values in numeric form.

As `print_vals` calls the various subroutines such as `volume` and `tone`, it passes the `GET` argument of `-1`, causing the appropriate sound chip register to be read.

Program 8-36. `printval.c`

```
# define GET    -1

print_vals(){
    char str[128];

    sprintf(str,"[0][%s!%s%5d%5d%5d!%s%5d%5d%5d!%s%5u %s%2d %s%2d][ OK ]",
        "Channel      A      B      C",
        "Volume: ",volume(0,GET),volume(1,GET),volume(2,GET),
        "Tone:      ",tone(0,GET),tone(1,GET),tone(2,GET),
        "Period:   ",period(GET),
        "Noise:    ",noise(GET),
        "Shape:    ",shape(GET)
    );
    form_alert(0,str);
}
```

Generating Rhythms with the Clock

The next few functions show how you can produce rhythms and noises with the ST's interval timing mechanism. In the beginning of this chapter, we set the interval variable to 20 milliseconds, causing GEM to send multi a message every 1/50 second.

Thus, 50 times a second, multi gets a message and calls the `clock_ticks` function, which checks the value of the `cur_rhythm` variable that is set by the `do_rhythm` function. If the `cur_rhythm` variable is not 0, the percussion subroutine is called to make sounds according to a specified pattern in order to produce rhythms.

The `drums.h` header file. The rhythm structure consists of a duration and an instrument. The instrument names are defined in the `drums.h` header file, Program 8-37. We define 17 possible "instruments," which are actually sounds played in structured patterns, and assign an arbitrary value to each instrument. We'll use this value later to quickly identify a pattern.

Program 8-37. drums.h

```
# define BASS_DRUM      1
# define SNARE_DRUM    2
# define TOMTOMLOW     3
# define TOMTOMMED     4
# define TOMTOMHIGH    5
# define CYMBALS       6
# define BLOCKLOW      7
# define BLOCKMED      8
# define BLOCKHIGH     9
# define BELLOW       10
# define BELLMED      11
# define BELLHIGH     12
# define SILENCE      13
# define BRUSHES      14
# define GUNSHOT      15
# define EXPLOSION    16
# define BROKEN_GLASS 17
```

```
struct rhythm {
    char duration;          /* how many 50ths of a second */
    char instrument;
};
```

The do_rhythm and do_effects functions. When the user selects Rhythms from the options pull-down menu, the DboDo_main_menu function calls do_rhythm, Program 8-38. For our purpose here, which is to illustrate how to generate these rhythms, we use a somewhat simple routine that must run through the complete list of rhythms before it ends. Once the user selects the Rhythms option, a sound pattern begins and continues until he or she selects the option again, which produces another sound. The user cycles through all seven rhythm patterns we've defined in order to return to silence.

First, we describe one cycle of seven different rhythm arrays, specifying the duration and instrument for each note. Except for the last array, battle, the rhythm cycles will repeat until the user selects Rhythms again. Each time the user selects Rhythms and do_rhythms is called, the rhythm number stored in *x* is incremented and the global variable *cur_rhythm* is set to the next rhythm. As long as *cur_rhythm* is not set to 0, the percussion subroutine will be called to play the sound pattern.

Selecting the Effects menu option will produce only one result: a series of battle noises. The do_effects subroutine sets the *cur_rhythm* variable to the battle rhythm array. Note the special value 0xff at the end of the array; it will be used to stop the battle sounds from being repeated.

Program 8-38. dorhythm.c

```
# include <drums.h>

struct rhythm rocknroll[] = {
    12, SNARE_DRUM, 12, BASS_DRUM, 12, BASS_DRUM, 12, BRUSHES,
    0, 0
};
struct rhythm justbass[] = {
    20, BASS_DRUM,
    10, BASS_DRUM,
```



```

static int x      = 0;

rhythm_index = 0;
switch(x++){
    case 0:
        cur_rhythm = rocknroll;
        break;
    case 1:
        cur_rhythm = justbass;
        break;
    case 2:
        cur_rhythm = justsnare;
        break;
    case 3:
        cur_rhythm = justbrush;
        break;
    case 4:
        cur_rhythm = bell_blocks;
        break;
    case 5:
        cur_rhythm = tomtoms;
        break;
    default:
        cur_rhythm = 0;
        x = 0;
        break;
}
}
do_effects(){
    rhythm_index = 0;
    cur_rhythm = battle;
}

```

The clock_ticks Function

The `cur_rhythm` variable is used by the `clock_ticks` function, Program 8-39, which is called by `multi` 50 times a second. The calling is controlled by the interval variable initialized in the `config.c` file.

If the `cur_rhythm` variable is 0, `clock_ticks` returns without making a sound.

As soon as the user selects the Rhythm option, `do_rhythm` sets the `cur_rhythm` variable to one of its rhythm arrays. `clock_ticks` finds that the variable does not equal 0, and sets the `time_count` variable to the duration of the first instrument in the specified rhythm array. The percussion function is called to play the sound, the `rhythm_index` is advanced to point to the next note, and `time_count` is decremented by 1. Now, every 1/50 second, `clock_ticks` finds that `cur_rhythm` is not 0, and decrements `time_count` and returns until `clock_ticks` has been called the number of times specified in the duration. When `time_count` reaches 0, we play the note, move to the next duration and instrument in the array, and repeat the process.

For the Effects menu item, we only play the sound pattern once instead of repeating it, as we do for the other rhythms. `clock_ticks` looks for the special duration value of 0xff and returns if it finds this value.

Program 8-39. clocktic.c

```
#include <drums.h>

clock_ticks(whand,vw)
int whand, vw;{

    static int time_count = 0;
    extern struct rhythm *cur_rhythm;
    extern int rhythm_index;

    if( cur_rhythm == 0 )
        return;
    if( --time_count > 0 )
        return;
    time_count = cur_rhythm[rhythm_index].duration;
    percussion( cur_rhythm[rhythm_index].instrument );
    rhythm_index++;
    if( cur_rhythm[rhythm_index].duration == 0 )
        rhythm_index = 0;
    if( cur_rhythm[rhythm_index].duration == 0xff )
        cur_rhythm = 0;
}
```

The percussion and bellblock functions. The rhythmic sound patterns are made by the percussion function, Program 8-40, which sets the sound chip registers and then sets the shape register to sound the note, as explained in the discussion of the play_note function. By experimenting with the different sounds caused by different register settings, we selected the ones most similar to the instruments we want to synthesize.

Some of the instruments are easy to imitate, so the register value is set for the tone subroutine to use.

The sounds for bells, block, and tomtoms are available in high, medium, and low pitches. To efficiently program the pitches, we take advantage of the ability of the C switch statement to provide multiple entry points to the same code. In other words, to make a low tomtom sound, the code is entered at

case TOMTOMLOW

and tone_val is incremented by 800. Then execution continues to TOMTOMMED, where another 400 is added, and then to TOMTOMHIGH, where 2047 is added. The resulting value, 3247, produces the low pitch. For TOMTOMMED, only the 400 value is added to 2047, resulting in a higher pitch. For TOMTOMHIGH, only 2047 is passed to tone to produce the highest pitch.

percussion calls the bellblock function, Program 8-41, to sound the woodblock and bell sounds, which are the same except that the woodblock has a shorter envelope period. Because they are the same sound, we use the same routine to generate them.

The arguments for bellblock are the tone value and envelope period, with which the subroutine sets the registers.

Program 8-40. percussn.c

```
# include <drums.h>

percussion(instrument)
int instrument;{

    int tone_val = 0;

    volume(0,0);
    volume(1,0);
    volume(2,0);
    switch( instrument ){
        case BASS_DRUM:
            period(8208);
            mode_bit( 0, 1 );
            mode_bit( 1, 1 );
            mode_bit( 2, 1 );
            noise_enable(0,1);
            noise_enable(1,0);
            noise_enable(2,0);
            tone(0,4092); tone_enable(0,1);
            tone(1,4092); tone_enable(1,1);
            tone(2,4092); tone_enable(2,1);
            break;
        case SNARE_DRUM:
            mode_bit( 0, 1 );
            mode_bit( 1, 1 );
            mode_bit( 2, 1 );
            period(2000);
            noise_enable(0,1);
            noise_enable(1,1);
            noise_enable(2,1);
            noise(4);
            tone_enable(0,0);
            tone_enable(1,0);
            tone_enable(2,0);
            break;
        case TOMTOMLOW:
            tone_val += 800;
        case TOMTOMMED:
            tone_val += 400;
        case TOMTOMHIGH:
            tone_val += 2047;
            period(8208);
            mode_bit( 0, 1 );
            mode_bit( 1, 1 );
            mode_bit( 2, 1 );
            noise_enable(0,1);
            noise_enable(1,0);
            noise_enable(2,0);
            tone(0,tone_val); tone_enable(0,1);
            tone(1,tone_val); tone_enable(1,1);
            tone(2,tone_val); tone_enable(2,1);
            break;
        case CYMBALS:
            break;
        case BLOCKLOW:
            tone_val += 8;
        case BLOCKMED:
            tone_val += 4;
        case BLOCKHIGH:
            tone_val += 60;
            bellblock(tone_val,752);
            break;
        case BELLOW:
            tone_val += 8;
    }
}
```

CHAPTER 8

```

case BELLMED:
    tone_val += 4;
case BELLHIGH:
    tone_val += 60;
    bellblock(tone_val, 9632);
    break;
case SILENCE:
    break;
case BRUSHES:
    mode_bit( 0, 1 );
    mode_bit( 1, 0 );
    mode_bit( 2, 0 );
    period(8000);
    noise(0);
    noise_enable(0,1);
    noise_enable(1,0);
    noise_enable(2,0);
    tone_enable(0,0);
    tone_enable(1,0);
    tone_enable(2,0);
    break;
case GUNSHOT:
    noise(15);
    noise_enable(0,1);
    noise_enable(1,1);
    noise_enable(2,1);
    tone_enable(0,0);
    tone_enable(1,0);
    tone_enable(2,0);
    mode_bit( 0, 1 );
    mode_bit( 1, 1 );
    mode_bit( 2, 1 );
    period(8192);
    break;
case EXPLOSION:
    noise(0);
    noise_enable(0,1);
    noise_enable(1,1);
    noise_enable(2,1);
    tone_enable(0,0);
    tone_enable(1,0);
    tone_enable(2,0);
    mode_bit( 0, 1 );
    mode_bit( 1, 1 );
    mode_bit( 2, 1 );
    period(28672);
    break;
case BROKEN_GLASS:
    noise_enable(0,0);
    noise_enable(1,0);
    noise_enable(2,0);
    tone_enable(0,1);
    tone_enable(1,0);
    tone_enable(2,0);
    mode_bit( 0, 1 );
    mode_bit( 1, 1 );
    mode_bit( 2, 1 );
    period(8192);
    break;
    }
shape(9);
}

```


Program 8-41. bellblok.c

```

bellblock(tone_val,p)
int tone_val, p){

    mode_bit( 0, 1 );
    mode_bit( 1, 1 );
    mode_bit( 2, 1 );
    noise_enable(0,0);
    noise_enable(1,0);
    noise_enable(2,0);
    tone_enable(0,1);
    tone_enable(1,1);
    tone_enable(2,1);
    period(p);
    tone(0,tone_val);
    tone(1,tone_val);
    tone(2,tone_val);
}

```

The Music

After each of these functions has been compiled, they are linked all together, when using the *Atari ST Software Developer's Kit* linker, with the linkit.bat batch file, Program 8-42, which reads a list of arguments from the link.arg file. The link.arg file, Program 8-43, lists all the programs and files constructed in this chapter; they appear in uppercase.

Note that one of the files listed is env.a, which is in the library of functions constructed in Chapter 2. Those files need to be linked for this application to work.

After constructing and compiling the files in the following listings, you can click on batch.ttp on the desktop and give "linkit" for the argument. After linking everything, and renaming the a.prg file to noise.prg, the program will wait for you to press a key.

Program 8-42. linkit.bat

```

c:\bin\link68 [undefined,symbols,command[link.arg]]
c:\bin\relmod a
c:\bin\rm a.68k
c:\bin\wait

```

Program 8-43. link.arg

```

a.68k=c:gemstart.o,main.o,
CONFIG.O,DOMENU.O,DOIT.O,JUSTDRAW.O,SLIDERS.O,DOMNMENU.O,
SHOWKEYS.O,FILLBOX.O,DOWHITE.O,DOBLACK.O,OPENDATA.O,
GOTKEY.O,DORHYTHM.O,CLOCKTIC.O,PERCUSSN.O,BELLBLOK.O,
PRINTVAL.O,SAVSTATE.O,DOSLIDE.O,SLSET.O,SLIDEVAL.O,MODEBITS.O,
WHICHONE.O,BLDSLIDE.O,ALLSLIDE.O,TOGGLES.O,RADBUTN.O,
SELECTON.O,ADDSLIDE.O,SETSLIDE.O,BLDTREE.O,VOLUME.O,TONE.O,
NOISE.O,TONEENAB.O,NOISENAB.O,PERIOD.O,SHAPE.O,PLAYNOTE.O,
env.a,vd:bind,vdidata.o,gemlib,aesbind,osbind,libf

```




9 A Debugging Aid



9 A Debugging Aid

■ This chapter will discuss programs that misbehave. Specifically we'll discuss programs that bomb—programs that produce fancy bomb icons in the middle of your screen—and (if you're lucky) return to the desktop, or (if you're not lucky) hang up the system. We'll talk about why they behave this way, and how to find and fix the problems that cause program crashes. To aid in debugging programs, a desk accessory will be presented that points out exactly where the program crashed by listing the names of all the functions that led up to the crash, and their arguments. It will also show the contents of all of the registers at the time of the crash, and allow you to page through a disassembled listing of the program, so you can see exactly what the problem was.

Exception Handling on the 68000

To explain program crashes, it is necessary to talk about the Central Processing Unit (CPU) in the ST: the M68000 CPU, or 68000, for short. The 68000 has a mechanism for handling severe program errors. This mechanism is called *exception handling*, and the severe errors that cause exception handling to be necessary are called exceptions.

Exceptions occur when the programmer inadvertently asks the 68000 to do something it cannot do. The exceptions that we're concerned with are Bus Errors, Address Errors, Illegal Instructions, Dividing by Zero, Indexing Errors (when protected by the CHK instruction), Overflow Errors (when protected by the TRAPV instruction), and Privilege Violations.

Bus errors. Bus Errors occur when the programmer references memory that does not exist, or is protected from being referenced. The ST has a Memory Management Unit that enables it to protect memory from being referenced by normal user programs to prevent the operating system's memory from accidentally being clobbered. Certain addresses in the lower part of the ST's memory can only be accessed by a user's program when it is in Supervisor mode, which is the mode that TOS runs in. If a normal program tries to read or alter this memory, a Bus Error results, and two bomb icons appear on the screen. To access this memory, a User-mode program can ask TOS to temporarily put it in Supervisor mode while it references the memory.

Address Errors. Address Errors occur when the 68000 is asked to reference a 2- or 4-byte object (an integer or a long integer) on an odd byte boundary. The 68000 requires that all references to integers or long integers be on

even boundaries. If they are not, three bomb icons appear on the screen.

Illegal Instructions. Illegal Instruction Errors happen when an instruction is encountered that is not in the 68000's instruction set. Since 68000 instructions are all at least 16 bits long, there are 65,536 possible instructions. The 68000 only recognizes a little over 1000 of these possible instructions, so the odds are good that if your program tries to execute a subroutine that has been damaged by accidentally writing data in it, you'll see four bomb icons on your screen. There is a special instruction in the 68000's instruction set called *illegal*. It can be placed where the program should never execute, to catch errors that would be difficult to find otherwise.

Zero Divide. Zero Divide Errors happen when a program tries to divide something by zero. In mathematics and in programming, the result would be undefined. The 68000 will generate a Zero Divide exception when this occurs. But you are likely never to see five bomb icons on the screen, since TOS does not consider this to be a serious error and just returns to the program, letting the result of the division be random garbage. We will show how to detect Zero Divide Errors without interfering with the execution of the program that has them.

CHK Instruction (Indexing Errors). Indexing Errors are caused by a special instruction in the 68000 called the CHK instruction. Some compilers (and some assembly language programmers) use the CHK instruction to make sure that indexes into arrays are never negative, and are never bigger than the size of the array. If they are, the the CHK instruction causes an Indexing Error exception, and six bomb icons appear on the screen.

TRAPV Instruction (Overflow). The Overflow Error is also caused by a special instruction, the TRAPV instruction. If the program tries to add two numbers whose result is too big to store, then an overflow is said to occur. If the compiler (or assembly language programmer) puts a TRAPV instruction after the ADD instruction, then seven bomb icons will appear on the screen whenever the ADD causes overflow.

Privilege Violations. Last, Privilege Violations occur when the program is in User mode and tries to execute an instruction that is only allowed in Supervisor mode, such as RESET or STOP. Eight bombs will appear if this ever happens.

A Desk Accessory for Catching Bugs

When TOS puts the bomb icons on the screen, it also saves some information about the program in a safe place. This place is not cleared when you push the RESET button, and can thus be examined after a crash. The 68000's registers are stored there, along with the last 32 bytes of the Supervisor mode stack, which contains the Program Counter (the address at which the program bombed) and some other information about the crash.

Unfortunately, the program itself does not survive in memory for us to look at, and the information about where the program was loaded in memory

is also lost. This makes the information saved by TOS almost useless as an aid in debugging the program.

We can make the information useful by arranging for some code of our own to be executed whenever an exception occurs, but before TOS prints the bomb icons. This code will save the information needed to debug the program, and then let TOS blow up the program.

The desk accessory being built in this chapter can be thought of as two programs. One half is concerned with catching the exceptions and saving the critical information about the crash. The other half is concerned with displaying the information to the user. We'll describe the first half of the program first: how to catch the exceptions and save the data.

The 68000 assigns numbers to its exceptions. TOS uses the exception number to determine how many bombs should be printed on the screen. Thus there are two bombs for Bus Error, which is exception number two. (Exception number one is system reset, which we don't consider to be a programming error.)

The 68000 has an array of pointers in low memory that point to subroutines that are called whenever an exception occurs. TOS sets these pointers at boot time, so that when a Bus Error occurs, the 68000 jumps to the TOS routine to handle Bus Errors. TOS provides a special function called Setexc (Set exception) that allows us to replace an exception pointer with one of our own, thus causing the 68000 to call our subroutine whenever the exception occurs.

The Setexc function takes two arguments: an exception number (for example, 2 for Bus Error), and a function address. It returns the old address, so we can store it somewhere and replace it when we no longer need to intercept the interrupt ourselves.

The configac.c File

The config file, Program 9-1, for this accessory is much the same as that of the shell. The changes are the usual ones involving the name of the program, and the size of the initial window is a little bigger to accommodate the information it needs to hold.

Program 9-1. configac.c

```
# include <gemdefs.h>

char *wind_name           = " Bombsite! ";

# ifdef USE_RCS
char *resource             = "BOMBSITE.RSC";
# else
char *resource             = 0;
# endif USE_RCS

char *access_name         = " Bombsite! ";
int i_am_accessory        = 1;
int sx                    = 20;    /* small window size */
int sy                    = 30;
int sw                    = 350;
```


CHAPTER 9

```
int sh          = 100;
int slv         = 0;    /* small window vertical slider pos */
int slh         = 0;    /* small window horizontal slider pos */
int svb         = 1;    /* small window vertical slider size */
int shs         = 1;    /* small window horizontal slider size */
int min_wide    = 20;
int min_high    = 20;
int interval    = 0;
int events = MU_MESAG | MU_KEYBD;
```

The open_data Function

The desk accessory that will help in debugging is called “Bombsite!” and it uses Setexc in its open_data function, Program 9-2.

The routines h_bus_err, h_addr_err, h_illegal_err, h_zerodiv_err, h_chk_err, h_trapv_err, and h_priv_err are the routines that will handle the exceptions. The pointers, olderr2 (and olderr3, and so on), will hold the old pointers that we are replacing.

The open_data function is called whenever the accessory is reopened after an application program is run. If it were allowed to execute twice, then the “olderr” values would be clobbered, so we keep a static variable called only_once which will be set to 1 when open_data has already been called. Then open_data can just return if it finds it has been called once already.

The set_top function is called by opendata to find out how much memory this ST has. This value will be used later to make sure that our accessory does not try to reference memory that is not there (it will be examining another program, so the normal precaution of only using its own data will not be sufficient to prevent errors).

Then open_data calls Setexc for each exception we wish to retarget to our own routines.

Program 9-2. opendata.c.

```
#include <osbind.h>

open_data(file,whand,vw)
char *file;
int whand, vw;{

    static int only_once    = 0;
    extern int h_bus_err(), h_addr_err(), h_illegal_err();
    extern int h_zerodiv_err(), h_chk_err(), h_trapv_err(), h_priv_err();
    extern int (*olderr2)();
    extern int (*olderr3)();
    extern int (*olderr4)();
    extern int (*olderr5)();
    extern int (*olderr6)();
    extern int (*olderr7)();
    extern int (*olderr8)();

    if( only_once ){
        return;
    }
    only_once = 1;
    set_top();
    olderr2 = Setexc(2,h_bus_err);
```



```

olderr3 = Setexc(3,h_addr_err);
olderr4 = Setexc(4,h_illegal_err);
olderr5 = Setexc(5,h_zerodiv_err);
olderr6 = Setexc(6,h_chk_err);
olderr7 = Setexc(7,h_trapv_err);
olderr8 = Setexc(8,h_priv_err);
}

```

The set_top Function

The set_top function, Program 9-3, is responsible for setting the phystop variable, which holds the highest RAM address (the *top* of physical memory). It calls a special routine called getlong which is able to read any address in RAM (even protected memory) because it runs in Supervisor mode (you'll see how it does that later). The address PHYSTOP is where TOS stores the value we want. It is in protected memory, and if getlong isn't used to get it, it could cause a Bus Error exception, and crash. If getlong returns a value for phystop that is less than 256K (hexadecimal 0x40000), then we assume that something is very wrong, and we set it to -1 (all of possible memory) and continue, in the hope that we can help the programmer discover the problem. 256K is the smallest ST made, so this value should be reasonable.

Program 9-3. settop.c

```

# define PHYSTOP      0x042eL

set_top(){
    extern long int phystop;

    phystop = getlong(PHYSTOP);
    if( phystop < 0x40000 )
        phystop = -1;
}

```

The errors.c File

Now that the exception array contains pointers to our subroutines, we can look at what happens when an exception occurs. Since the exception-handling routines are all similar, we group them into one file, called errors.c.

The error-handling routines need to do things that are not easily done from C, like saving and restoring all of the registers, and jumping to the TOS exception handler without disturbing any registers. Since the amount of work we need to do in assembler is small, we use the *asm* feature of C to include in-line assembly instructions into the code at this point.

The h_bus_err routine is used as an example, since all of the other routines follow the same logic. The first line uses the *move multiple* instruction to copy all 16 68000 registers into the array saveregs. The second line calls the function get_trace, which will save the critical data from the program that has caused the exception. The third line restores all 16 registers from the saveregs array. We need to save and restore the registers, because the get_trace function will be changing them, and we want them in their original state for the next few lines.

The next three lines implement an assembly language trick that allows the accessory to jump to TOS at the place where TOS would have been entered if we hadn't intercepted the exception. To TOS, we wish the state of the machine to be exactly the same as if we had never interfered. To do this, we must remove from the stack anything that our subroutine placed there. When `h_bus_err` was called, it placed register A6 on the stack as part of the normal C subroutine call sequence. A6 is also called the "frame pointer," or in DRI assembly language, R14. The UNLK instruction removes it, and puts the stack back in the state it was in when `h_bus_err` was called. Then `h_bus_err` pushes the old exception pointer onto the stack and executes a RTS instruction, which pops it back off the stack and jumps to it (this is the trick we use to jump through a pointer without disturbing any registers).

All of the other exception-handling routines are similar. They differ only in the old pointer they jump to, and in the way they call `get_trace`. Bus Errors and Address Errors have extra information on the stack that the other exceptions don't have. This information includes the address being referenced that caused the exception to occur (such as the address that was out of bounds or odd). `Get_trace` is given an initial argument of 1 to indicate that this extra information is there, and 0 if it is not there. In most exceptions, the Program Counter (the address of the instruction that caused the exception) is pointing at the instruction AFTER the one that caused the error. In the case of Illegal Instructions, or Privilege Violations, however, the address is pointing right at the offending instruction. The second argument to `get_trace` is 0 if the Program Counter points beyond the instruction, and 1 if it points at it.

The last argument to `get_trace` is the name of the exception.

Program 9-4. errors.c

```
int (%olderr2)();
int (%olderr3)();
int (%olderr4)();
int (%olderr5)();
int (%olderr6)();
int (%olderr7)();
int (%olderr8)();

long int saveregs[16];

h_bus_err(arg)
short int %arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 1, 0, "Bus Error" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14");                               /* pop off fp */
    asm("move.l _olderr2,-(sp)");                   /* move old trap address onto stack */
    asm("rts");                                     /* jump to old trap address */
}

h_addr_err(arg)
short int %arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 1, 0, "Address Error" );
```

```

    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr3,-(sp)");
    asm("rts");
}

h_illegal_err(arg)
short int $arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 0, 1, "Illegal Instruction" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr4,-(sp)");
    asm("rts");
}

h_zerodiv_err(arg)
short int $arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 0, 0, "Zero Divide" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr5,-(sp)");
    asm("rts");
}

h_chk_err(arg)
short int $arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 0, 0, "Chk Instruction" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr6,-(sp)");
    asm("rts");
}

h_trapv_err(arg)
short int $arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 0, 0, "Trapv Instruction" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr7,-(sp)");
    asm("rts");
}

h_priv_err(arg)
short int $arg;{

    asm("movem.l d0-d7/a0-a7,_saveregs");
    get_trace( 0, 1, "Privilege Violation" );
    asm("movem.l _saveregs,d0-d7/a0-a7");
    asm("unlk R14"); /* pop off fp */
    asm("move.l _olderr8,-(sp)");
    asm("rts");
}

```

The get_trace Function

The `get_trace` function, Program 9-5, looks at the stack to see where the defective program has been. Because each time a function is called in C the return address and frame pointer are pushed on the stack, the stack has a record of every function that was called before the crash. All `get_trace` needs to do is to read the stack and decipher the information. (Since all the exception pointers now end up calling `get_trace`, we need to protect ourselves from exceptions

while within `get_trace`. If one were to occur, then `get_trace` would be called from within itself, which could lead to confusion. The static integer `already_in` causes `get_trace` to return immediately if it is ever called from within itself by accident.)

The address of the top of the stack is the address of the last thing pushed on the stack. In this case, the last thing pushed was the first argument to `get_trace`, the integer `bus_or_addr`.

The pointer `short_ptr` is set to point at the top of the stack, and it is then used to get the frame pointer and the Program Counter of the crashing program into the variables `fp` and `pc`. If there was extra information on the stack (because the exception was a Bus Error or an Address Error) then the `pc` (Program Counter) will be found 11 short integers up the stack. If no extra information was on the stack, then the `pc` will be found 7 short integers up the stack. (Refer to Figure 9-1 to see where these numbers come from.)

If neither the `fp` or the `pc` could be found (`getlong` returns 0 if it can't get the data), then `get_trace` gives up and returns. If the `pc` was found, and it points to memory that exists, then the `get_dis` routine is called to disassemble the area around the crash location, producing an assembly listing of the program at that point.

Note: All of the code that deals with disassembling the offending program is placed between *ifdef* and *endif* statements. This allows you to build the stack trace and bomb info parts of the program, and leave the disassembler until later. The disassembler is in the next chapter. If you're typing this program in, you'll then get immediate use out of the program, and not have to type in the disassembler.

Program 9-5. `gettrace.c`

```
# include <debug.h>
# include <document.h>
# define TRACE_NUM      32
# define NUM_ARGS       32

long int real_pcs[TRACE_NUM+1];
int numpcs              = 0;

pc_compar(a,b)
long int *a, *b;{

    return( *a - *b );
}

struct trc savtrc[TRACE_NUM];

get_trace(bus_or_addr, exact_pc, err_str)
short int bus_or_addr, exact_pc;
char *err_str){

    long int *fp, *fp_addr, **long_ptr, getlong();
    short int *pc, *short_ptr, getshort();
    int i, j;
    extern long int proc_pc;
```



```

extern short int *get_realpc();
static int already_in = 0;

if( already_in++ ){      /* Guard against errors while in get_trace */
    return;
}

/*
** Get the frame pointer which is stacked before the arguments
*/
short_ptr = &bus_or_addr;
long_ptr = (long int **)&short_ptr[-4];
fp = getlong(&long_ptr);
if( fp == 0 ){
    already_in = 0;
    return;
}

if( bus_or_addr )
    pc = (short int *) getlong(&short_ptr[11]);
else
    pc = (short int *) getlong(&short_ptr[7]);
if( pc == 0 ){
    already_in = 0;
    return;
}

proc_pc = (long int) pc;

# ifdef HAS_DISASSEMBLY
if( getshort(proc_pc) )
    get_dis( N_LINES, exact_pc );
# endif HAS_DISASSEMBLY

for( i = 0; fp && i < TRACE_NUM-1; i++ ){
    savtrc[i].fp = fp;
    savtrc[i].ret_pc = pc;
    pc = (short int *)getlong(fp+1);
    if( i )
        savtrc[i].real_pc = get_realpc(pc);
    else
        savtrc[i].real_pc = savtrc[i].ret_pc - !exact_pc;
    savtrc[i].num_args = get_args(fp,pc,1,savtrc[i].args);
    fp_addr = fp;
    fp = (long int *)getlong(fp_addr);
    if( fp == fp_addr )
        break;
}

savtrc[i].fp = 0;
numpcs = 0;
for( j = 0; j < i; j++ ){
    if( savtrc[j].real_pc )
        real_pcs[numpcs++] = savtrc[j].real_pc;
}
qsort( real_pcs, numpcs, sizeof( real_pcs[0] ), pc_compar );
already_in = 0;
}

```

The stack. At several points in this program we'll be very cautious about pointers found on the stack. Since the program we are examining is known to be misbehaving, it may have destroyed part or all of its stack, or its program code. This is why `get_trace` checks `fp` and `pc` so carefully, and why it checks to make sure that the `pc` points to valid memory before calling `get_dis`.

At this point, the picture of the stack structure in Figure 9-1 will be worth a thousand words.

Figure 9-1. The Stack Trace Structure

&short_ptr[-4] →	low word of old Frame Pointer
	high word of old Frame Pointer
	low word of Return Address
	high word of Return Address
&bus_or_addr →	bus_or_addr
	exact_pc
	low word of err_str
	high word of err_str
	low word of h_bus_err fp
	high word of h_bus_err fp
	function code
	low word of access address
	high word of access address
	instruction that failed
	status register
	low word of program counter
&short_ptr[11] →	high word of program counter

In the C function calling sequence, the current frame pointer always points at the saved copy of the previous frame pointer. In this way, the frame pointers form a linked list of pointers that we can follow. In normal execution (for instance, in the absence of exceptions) the return address of a function is always found immediately after the frame pointer. This can be seen in the top four items in the illustration of the stack frame. In the exception frame, the program counter and the old frame pointer are separated by the status register. If the exception was a Bus Error or an Address Error, they are separated by a

function code, the address that caused the problem, and the first word of the instruction that failed.

To follow the linked list of stack frames, `get_trace` loops until the value it gets for `fp` is invalid (zero), or the space reserved for storing stack frames is used up (when `i` is greater than or equal to `TRACE_NUM-1`). The `fp` is put into `savtrc[i].fp`, and the `pc` is put into `savtrc[i].ret_pc`. Notice that this `pc` is the return address, not the address of the function.

`Get_trace` then gets the next `pc` by using `getlong` to fetch the long integer just above the current frame pointer (`fp+1`). If this is the first stack frame (if `i = 0`) then the `get_realpc` function will not be able to find the real address of the start of the function, since it needs the previous stack frame to work from. (We will discuss `get_realpc` in a moment. Its purpose is to guarantee that we know the current function's name by looking at how the previous subroutine called it.) In the case of the first stack frame, we set `savtrc[i].real_pc` to the best guess we have of the address: the current return address (minus one word if the exception was not an Illegal Instruction or a Privilege Violation).

The arguments of the function are collected off the stack by the `get_args` routine, which returns the number of arguments found. Last, the new frame pointer is fetched (easily, since the current `fp` always points to the next one). In case the stack was damaged, we check to make sure that it is different from the old one to prevent loops.

After the loop, we collect all of the `real_pc` entries, and save them in an array called `real_pcs`, which we sort using `qsort`. This sorted list of subroutine addresses will be used by a very tricky routine (called `get_base`) which will attempt to reconstruct the program's load address with the symbol table from the program and the sorted list in `real_pcs`.

The `get_real` Function

The stack trace that `get_trace` builds is just a list of addresses and function arguments. It's difficult to read a list of numbers and try to figure out which functions the list refers to. The purpose of the `get_realpc` function, Program 9-6, is to aid later functions (`get_base` and `get_name`) in putting real names in the trace instead of just numbers.

TOS does not save an important ingredient needed to do this: the load address of the program. The names of all the functions are stored in the symbol table of the program on the disk that the user double-clicked on. These names have addresses associated with them, but the addresses are all relative to the beginning of the file. When TOS loads a program, it picks an address in memory where the program will reside, and then it adds that address to each address in the symbol table. When `get_trace` finds an address, it is the result of that addition. In order to compare the address of a subroutine name with that of an address, the load address must be added to the address found in the symbol table. The first step in finding the load address is to collect all of the known subroutine entry points possible.

The `get_realpc` function tries to find the address of the beginning of a subroutine by looking at the way that subroutine was called. The `get_realpc` subroutine is passed the address that the target subroutine will return to when it is done. That address will point to the instruction just after the instruction that called the target subroutine.

There are five possible ways that a C routine can call another C routine (there are actually more ways to do it, but C compilers generally limit themselves to these five). They are:

short branch to subroutine	(bsr.b)
long branch to subroutine	(bsr.w)
short jump to subroutine	(jsr.w)
long jump to subroutine	(jsr.l)
indirect jump to subroutine	(jsr (A0))

Each of these instructions calculates the address to jump to in a different way. The first one stores a number in the instruction itself; that number is added to the address of next instruction, and the result is where it jumps. The second one stores a similar number after the instruction, and adds to it the address of the next instruction to arrive at a target address. The third type of instruction stores a number after the instruction that is the address itself, and needs no addition to obtain a target address. The fourth type is like the third, but the number stored is 32 bits long instead of 16, so the entire addressing range of the 68000 is addressable using it.

The last type of jump is the indirect jump, and no numbers are stored after it. Instead, the target address is in a register.

In all but the last type of function call, `get_realpc` can examine the instruction that called the target subroutine, determine which type of instruction it was that did the call, and reconstruct the address that was called. This gives the address of the beginning of the subroutine that was called.

The `get_realpc` function starts by calculating the addresses of the possible subroutine call instructions. There are three possible instruction lengths: two bytes (address in `bret`), four bytes (address in `sret`), and six bytes (address in `lret`). The instructions found at these addresses are placed in `b_instr`, `s_instr`, and `l_instr`, respectively. If `get_realpc` is unable to get any of these, it assumes that the program is corrupted, and gives up.

If the `s_instr` word matches the BSRW pattern, then it must be a long branch to subroutine instruction. The offset is taken from the next word, and added to the address of the word following the instruction. This is done with the expression:

```
&sret[(offset>>1)+1]
```

The result is the address of the subroutine that the long branch to subroutine jumped to.

The short branch to subroutine is handled in a similar manner, but the offset is taken out of the low byte of the instruction itself.

The jump to subroutine instructions are simple, since the address is found immediately after the instruction, and no addition is necessary.

For the indirect jump case, there is nothing to be done, since there is no easy way to reconstruct the value in the register that was used for the jump. For this case `get_realpc` just returns 0, to indicate that it was unable to get a good value. Likewise, if none of the instructions matched a known instruction, then 0 is returned.

Program 9-6. `getreal.c`

```
# define BSR_LEN      0x00FF /* bsr instr. offset */
# define BSR_INSTR    0x6100 /* bsr instr. */
# define BSR_MASK     0xFF00 /* mask for bsr instr. */
# define BSRW_INSTR   0x6100 /* bsr word instr. */
# define BSRW_MASK    0xFFFF /* mask for bsr instr. */
# define JSRL_INSTR    0x4EB9 /* jsr abs long instr. */
# define JSRL_MASK     0xFFFF /* mask for jsr instr. */
# define JSRW_INSTR    0x4EB8 /* jsr abs short instr. */
# define JSRW_MASK     0xFFFF /* mask for jsr instr. */
# define JSRI_INSTR    0x4E90 /* jsr indirect instr. */
# define JSRI_MASK     0xFFFF /* mask for jsr instr. */

short int *
get_realpc(retadr)
short int *retadr; {

    int i;
    unsigned short l_instr, s_instr, b_instr;
    short int *shortp, getshort();
    long int getlong();
    char offset;
    short int *lret, *sret, *bret;

    lret = retadr-3;
    sret = retadr-2;
    bret = retadr-1;
    l_instr = getshort(lret);
    if( l_instr == 0 )
        return(0L);
    s_instr = getshort(sret);
    if( s_instr == 0 )
        return(0L);
    b_instr = getshort(bret);
    if( b_instr == 0 )
        return(0L);
    if ((s_instr & BSRW_MASK) == BSRW_INSTR) {
        offset = getshort(&sret[1]);
        retadr = &sret[(offset>>1)+1];
        return(retadr);
    }
    if ((b_instr & BSR_MASK) == BSR_INSTR) {
        offset = b_instr & BSR_LEN;
        retadr = &bret[(offset>>1)+1];
        return(retadr);
    }
    if ((l_instr & JSRL_MASK) == JSRL_INSTR) {
        retadr = getlong(&lret[1]);
        return(retadr);
    }
}
```

```

if ((s_instr & JSRW_MASK) == JSRW_INSTR) {
    retadr = (short int *) getshort(&sret[1]);
    return(retadr);
}
if ((b_instr & JSRI_MASK) == JSRI_INSTR) {
    return(0L);
}
return(0L);
}

```

The get_args Function

The get_args function, Program 9-7, uses a similar trick to find out how many arguments were passed to a subroutine.

When a function is called, the arguments to the function are pushed on the stack first; then the function is called. When the function returns, the instruction after the function call is usually an instruction that pops the arguments back off the stack. By examining this instruction, get_args can determine how many arguments were popped off, which usually correlates with how many arguments were pushed.

There are three instructions that are commonly used by C compilers to remove arguments from the stack. They are the ADDQ instruction, the ADDL instruction, and the LEA instruction. The ADDQ stores the number of arguments in the instruction itself, and get_args extracts that number into the variable nargs. If nargs was 0, then the 68000 interprets it as 8 (since adding 0 to something is useless), so get_args changes 0 to 8 on all ADDQ instructions.

The ADDL and LEA instructions simply store the number of arguments after the instruction, and it is a simple matter to read them into nargs.

If the instruction was a branch or a jump, then get_args assumes that the instruction that pops the stack is at the other end of the branch. It calculates the branch target address, and calls itself again to handle it. To avoid loops, the recurse argument insures that this is done only once, on the assumption that the code being read may have been damaged by the crash.

To complicate matters, the first argument is not pushed onto the stack, but merely moved onto the top of the stack, so functions that only have one argument have no code that cleans up the stack after the subroutine call. If there are no recognizable instructions after the function call, get_args sets nargs to the size of a long integer.

Since nargs is calculated in bytes, get_args divides it by two to get words, then adds one for the first argument, which was not popped. Then it loops nargs times, collecting each argument and putting it in the array args. Finally, it returns the number of arguments found.

Program 9-7. getargs.c

```

#define ADDQL_MASK      0xF1FF /* mask for addql instr. */
#define ADDQL_INSTR     0x50BF /* format of addql instr. */
#define ADDQL_SHIFT     9      /* shift count for addql */
#define ADDL_INSTR      0xDFFC /* addl instruction */
#define LEA_INSTR       0x4FEF /* format of lea instr. */
#define BRA_MASK        0xFF00 /* mask for bra instr. */

```

```

# define BRA_INSTR      0x6000 /* bra instr. */
# define BRA_LEN       0x00FF /* displ for bra instr. */
# define JMP_MASK      0xFFFF /* mask for jmp instr. */
# define JMP_INSTR     0x4EF9 /* jmp long abs instr. */

get_args(fp,retadr,recurse,args)
long int *fp;
short int *retadr;
int recurse;
short int *args;{

    int i;
    unsigned short instr;
    short int *shortp;
    char nargs;
    extern long int getlong();

    instr = getshort(retadr);
    if( instr == 0 )
        return;
    nargs = 0;
    if ((instr & BRA_MASK) == BRA_INSTR) {
        nargs = instr & BRA_LEN;
        if (nargs == 0)
            nargs = getshort(&retadr[1]);
        else if (nargs == -1)
            nargs = getlong(&retadr[1]);
        retadr = &retadr[nargs+1];
        if( recurse )
            get_args(fp,retadr,recurse-1,args);
        return;
    }
    if ((instr & JMP_MASK) == JMP_INSTR) {
        retadr = getlong(&retadr[1]);
        if( recurse )
            get_args(fp,retadr,recurse-1,args);
        return;
    }
    if ((instr & ADDQL_MASK) == ADDQL_INSTR) {
        nargs = (instr & ~ADDQL_MASK) >> ADDQL_SHIFT;
        if (nargs == 0)
            nargs = 8;
    }
    else if (instr == ADDL_INSTR){
        nargs = getlong(&retadr[1]);
    }
    else if (instr == LEA_INSTR){
        nargs = getshort(&retadr[1]);
    }
    else {
        nargs = sizeof(long);
    }
    nargs /= sizeof(short); /* convert to number of arguments */
    nargs++;
    fp += 2; /* step over linked fp and return address */
    shortp = fp;
    for( i = 0; i < nargs; i++){
        args[i] = getshort(shortp++);
    }
    return(nargs);
}

```

The getlong, getshort, and getbyte Functions

The getlong and getshort functions have been used extensively in the program so far to safely get values out of memory, no matter where they are. To do this, they need to do two things. One is to make sure that there is RAM at the location referenced. The other is to make sure the actual accessing is done in supervisor mode when accessing protected memory.

The getlong, getshort, and getbyte functions are all in one file, Program 9-8, for convenience, since they are all very similar.

Look at getshort as a typical example. The address passed is first compared to the pointer phystop, which was set when open_data called setup. If the address is beyond phystop, then getshort returns 0. (While experimenting with this program, you may want to know when this is occurring. There is an error message printed by show_form that will greatly aid in debugging any enhancements you make, but can be removed when you are content with the code.)

Getshort then sets the global Sadr to the address, and uses the macro Supexec (defined at the top of the file) to set Supervisor mode while the Getshort routine is called to fetch a short integer in supervisor mode. The result is masked off and returned.

Program 9-8. getlong.c

```
# include <osbind.h>
# define Supexec(x)    xbios(38,x)

long int bios(), xbios();

char *Cadr;
short int *Sadr;
long int *Ladr;
short int *phystop    = 0x40000;

Getbyte(){
    return(*Cadr);
}
short int
Getshort(){
    return(*Sadr);
}
long int
Getlong(){
    return(*Ladr);
}
getbyte(addr)
short int *adr;{

    char str[128];

    if( adr >= phystop ){
        sprintf(str,"Getbyte: adr(%X) >= phystop(%X)",adr,phystop);
        show_form(str);
        return(0);
    }
    if( adr >= phystop )
        return(0);
```



```

    Cadr = adr;
    return(Supexec(Getbyte) & 0xff);
}

short int
getshort(adr)
short int *adr;{

    char str[128];

    if( adr >= phystop ){
        sprintf(str,"Getshort: adr(%X) >= phystop(%X)",adr,phystop);
        show_form(str);
        return(0);
    }
    if( adr >= phystop )
        return(0);
    Sadr = adr;
    return(Supexec(Getshort) & 0xffff);
}

long int
getlong(adr)
long int *adr;{

    char str[128];

    if( adr >= phystop ){
        sprintf(str,"Getlong: adr(%X) >= phystop(%X)",adr,phystop);
        show_form(str);
        return(0);
    }
    if( adr >= phystop )
        return(0);
    Ladr = adr;
    return(Supexec(Getlong));
}

```

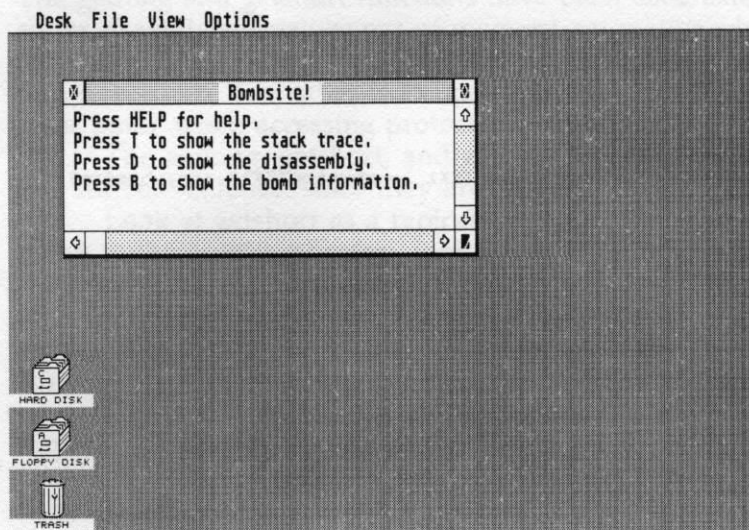
The doit Function

That ends the discussion of the first half of the program, the half that collects the data. The second half is concerned with presenting the data to the user, and starts with the `doit` function, Program 9-9.

The `doit` and `just_draw` functions put a simple menu on the screen that prompts the user to type T for a stack trace, or B for the bomb information. If the disassembler is included, then the D option is also displayed. The output of each of these commands will be shown in a separate window, just like the directory window in the shell program. To make this happen, the `dis_window` variable is used (just like `dir_window` in the shell) in the `show_info` function to indicate that a different routine is to be used to display the data (in this case, the `showwnd` routine).

The initial window looks like Figure 9-2.

Figure 9-2. The Initial Bombsite! Screen



Program 9-9. doit.c

```
# include <gemdefs.h>

int dis_window      = 0;

char *info[] = {
    "Press HELP for help.",
    "Press T to show the stack trace.",
    # ifdef HAS_DISASSEMBLY
    "Press D to show the disassembly.",
    # endif HAS_DISASSEMBLY
    "Press B to show the bomb information.",
    0
};

doit(whand,vw)
int whand, vw;{

    hide_mouse();
    clr_display(whand,vw);
    show_info(whand,vw);
    show_mouse();
}

show_info(whand,vw)
int whand, vw;{

    int x, y, w, h, i;
    extern int gl_wchar, gl_hchar;

    hide_mouse();
    if( whand == dis_window )
        showwnd(whand,vw);
    else {
        wind_get( whand, WF_WORKXYWH, &x, &y, &w, &h );
        for( i = 0; info[i]; i++ )
            v_gtext( vw, x+gl_wchar, y+(i+1)*gl_hchar, info[i] );
    }
}
```

```

    show_mouse();
}
just_draw(whand,x,y,w,h,vw)
int whand, x, y, w, h, vw;{

```

```

    hide_mouse();
    just_clear(whand,vw);
    show_info(whand,vw);
    show_mouse();
}

```

The got_key Function

When a key is pressed, got_key (Program 9-10) is called to handle it.

As in the Command Shell program, got_key returns 1 to indicate EXIT if the window that got the key was not the initial window.

If the key was the HELP key, then the familiar give_help function is called to lend assistance.

got_key then creates a string to be used by form_alert, requesting the user to select one of the options. After converting the character to uppercase ASCII, got_key selects a command from TRACE, BOMBINFO, or (optionally) DISASSEMBLE. If the key was not T, D, or B, then form_alert is called to ask in a different way.

Then a switch statement is used to call the appropriate routine (trace, disassem, or bombinfo) to format the data, and do_new_window (similar to the Command Shell's do_dir_window) to create a new window to display the data.

Program 9-10. gotkey.c

```

#include <document.h>
#include <debug.h>

#define RETURN          015
#define NEWLINE         012
#define CTRL_C          003
#define ESCAPE          033
#define BACKSPACE       010
#define HELP            0x6200
#define UNDO             0x6100

#define TRACE           1
#define BOMBINFO        2
#define DISASSEMBLE     3

got_key(ch,whand,vw)
int ch, whand, vw;{

    char s[128];
    int cmd;
    extern int highlight, dis_window;

    if( whand == dis_window )
        return(1);
    if( ch == HELP ){
        give_help(whand,vw);
        return(0);
    }

```

CHAPTER 9

```
# ifdef HAS_DISASSEMBLY
    sprintf(s,"%s%s",
        "[1][ Select one of these functions: ]",
        "[ TRACE ! BOMBINFO ! DISASSEM ]");
# else
    sprintf(s,"%s%s",
        "[1][ Select one of these functions: ]",
        "[ TRACE ! BOMBINFO ]");
# endif HAS_DISASSEMBLY
highlight = 0;
ch &= 0xdf; /* convert to upper case, 1 byte only */
if( ch == 'T' )
    cmd = TRACE;
else if( ch == 'D' )
    cmd = DISASSEMBLE;
else if( ch == 'B' )
    cmd = BOMBINFO;
else
    cmd = form_alert(1,s);
switch( cmd ){
    case TRACE:
        trace(whand,vw);
        do_new_window("Stack Trace",whand,vw,60,20);
        break;

    case BOMBINFO:
        bomb_info(whand,vw);
        do_new_window("Bomb Information",whand,vw,60,20);
        break;

    case DISASSEMBLE:
        highlight = 1;
        disassem(whand,vw,NLINES);
        do_new_window("Disassembly",whand,vw,65,24);
        break;
}
return(0);
}
```

The give_help Function

The give_help function should look familiar—Program 9-11.

Give_help puts up form_alert windows telling how to operate the program, explaining the trace, disassembly (if HAS_DISASSEMBLY was defined), and bomb information functions.

Program 9-11. givehelp.c

```
# include <debug.h>

give_help(whand,vw)
int whand, vw;{

    char str[256];

    sprintf(str,"[0][%s!%s!%s!%s!%s][ NEXT ]",
        "Type 'T' for a trace, showing",
        "which routines were called in",
        "what order, and the arguments",
```



```

        "to the routines, up until the",
        "program failed. ('Bombed')). "
    );
    form_alert(1,str);
    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "The program will prompt for a",
        "program name to read a symbol",
        "table from. Select the name ",
        "of the program that crashed. ",
        ""
    );
    form_alert(1,str);
# ifdef HAS_DISASSEMBLY
    sprintf(str,"[0][%s:%s:%s:%s][ NEXT ]",
        "Type 'D' for a disassembly of",
        "the ~200 lines of code around",
        "the point at which the faulty",
        "program crashed. The failure",
        "will be printed in bold face."
    );
    form_alert(1,str);
# endif HAS_DISASSEMBLY
    sprintf(str,"[0][%s:%s:%s:%s][ LAST ]",
        "Type 'B' for a register dump,",
        "showing the registers at the ",
        "time of the crash, including ",
        "the decoded Status Register, ",
        "and User Stack Pointer. "
    );
    form_alert(1,str);
}

```

The trace Function

When the user types T, `got_key` calls the trace function, Program 9-12. The trace subroutine is responsible for formatting and printing the information saved by `get_trace` in the first half of the program. As with the shell program, trace will put its data into the array `pl[]`, which will be displayed by `showwnd` when GEM sends a REDRAW message. Multi will catch the REDRAW message, pass it to `was_msg`, which calls `do_redraw`, which calls `just_draw`, which calls `show_info`, which calls `showwnd`.

Trace clears the `pl[]` array by putting zeros in the first byte of each line (thus making each line look like a null string to `v_gtext`). If there is no stack trace to print (for instance, `savtrc[0].fp` is still zero), then trace places a message into the `pl[]` array and returns. The message will be displayed when the REDRAW message is received.

Trace then calls `get_syms` to prompt for a filename. This file (presumably the name of the program which crashed) will be read, and its symbol table (the list of subroutine names and their addresses) extracted and sorted. Trace calls `get_base` to use this symbol table (and the list of sorted subroutine addresses collected by `get_trace`) to figure out where the program was loaded. Since the procedure used is not guaranteed to yield a unique load address (after all, `get_base` is trying to reconstruct information that has been thrown away), trace will present all of the possible interpretations, and the user can pick the one that looks right. In practice, if there are more than three functions in the

trace, `get_base` will find a unique load address, and only one stack trace will appear. The more data `get_base` has to work with, the better its guess will be.

To put the stack trace (or traces) into the `pl[]` array, trace calls the `bt` subroutine (short for `backtrace`). Before going into `bt`, a look at `get_syms` and `get_base` is in order.

Program 9-12. `trace.c`

```
# include <document.h>
# include <debug.h>

long int bases[256];
long int prog_base;

trace(whand,vw)
int whand, vw;{

    int x, next_line, num_bases;
    extern char pl[NLINES][NCHARS];
    extern int xlines;
    extern struct trc savtrc[];

    xlines = NLINES;
    for( x = 0; x < NLINES; x++ )
        pl[x][0] = 0;
    clr_disp(whand,vw);
    if( savtrc[0].fp == 0 ){
        sprintf(pl[0],"No program has bombed since");
        sprintf(pl[1],"this accessory has been loaded");
        sprintf(pl[2],"so there is no stack trace to");
        sprintf(pl[3],"display.");
        return;
    }
    get_syms(whand,vw);
    num_bases = get_base();
    next_line = 0;
    for( x = 0; x < num_bases; x++ )
        next_line = bt( next_line, bases[x] );
}
```

The `get_syms` Subroutine

The `get_syms` function, Program 9-13, uses the familiar `Dgetdrv`, `Dgetpath`, and `fsel_input` routines to put up a file selection window, just like the `Plot` program. Then it reads and processes the symbol table.

Each executable file on the ST begins with a file header, and each file header takes the form of the "head" structure declared in `get_syms`. `Get_syms` constructs a filename by combining the directory and the filename returned by `fsel_input`, and uses it to open the file with `Fopen`. It then reads the file header into the "head" structure. `Get_syms` is interested in the symbol table, which is stored in the file after the program code and data. To get to the symbol table, `get_syms` adds the size of the header to the size of the program code and data, and puts the result in `seek_ptr`. Then `Fseek` is called to "seek" to the beginning of the symbol table in the file. The number of symbols is calculated using the size of a symbol structure and the number of bytes in the symbol table (stored in `ssize` in the header). If there are too many symbols (unlikely), then an error

message is shown, and the number of symbols is adjusted by ignoring the symbols that don't fit.

Get_syms then loops, calling Fread to read each symbol into the array syms[]. Only TEXT symbols are stored; DATA symbols (and anything else) are ignored, since subroutine names are always TEXT symbols. The file is closed after the loop, and qsort is called to sort the symbols by their addresses. Finally, get_syms returns 1 to indicate success.

Program 9-13. getsyms.c

```
# include <osbind.h>
# include <debug.h>
# define CANCEL          0
# define OK              1

struct sym syms[256];

int count;

static compar(a,b)
struct sym *a, *b;{

    return( a->value - b->value );
}

# define TEXT 0x200

long int
get_syms(whand,vw)
int whand, vw;{

    int fd, x, button, drv, t_count;
    long int seek_ptr, ref;
    struct head {
        int magic;
        long tsize, dsize, bsize, ssize, zsize, entry;
        int reloc;
    } head;
    static char dir[256], file[256];
    char str[128], program[128], *p, *last_slash;

    if( dir[0] == 0 ){
        drv = Dgetdrv();
        Dgetpath(str,drv+1);
        sprintf( dir, "%c:%s\\%.s", drv+'A', str );
    }

    if( fsel_input(dir,file,&button) == 0 ){
        show_form("Error in file selection!No file selected");
        return(0);
    }

    if( button == CANCEL )
        return(0);
    strcpy(syms[0].name,"nameless");
    syms[0].value = 0;
    syms[0].type = 0;
    strcpy(str,dir);
    last_slash = str;
    for( p = str; *p; p++ )
        if( *p == '\\' )
            last_slash = p;
    *last_slash = 0;
    sprintf(program,"%s\\%.s",str,file);
```

```

fd = Fopen(program,0);
if( fd < 0 ){
    sprintf(str,"Can't open '%s'",program);
    show_form(str);
    return(0);
}

count = Fread(fd,(long int)sizeof(head),&head);
seek_ptr = sizeof(head)+head.tsize+head.dsize;
Fseek(seek_ptr,fd,0);
count = head.ssize / sizeof(syms[0]);
if( count > sizeof(syms) / sizeof(syms[0]) - 2 ){
    count = sizeof(syms) / sizeof(syms[0]) - 2;
    sprintf(str,"Too many symbols: using %d",count);
    show_form(str);
}

t_count = 0;
for( x = 1; x < count; x++ ){
    Fread(fd,(long int)sizeof(syms[0]),&syms[t_count]);
    if( syms[t_count].type & TEXT )
        t_count++;
}

count = t_count;
Fclose(fd);
qsort(syms,count,sizeof(syms[0]),compar);
return(1);
}

```

The get_base Function

Get_base is a deceptively simple routine (Program 9-14) that tries to figure out where the program was loaded, given only a list of subroutine addresses from the executable file (relative to zero) and a list of function addresses from the crashed program (relative to the unknown load address).

The idea is to compare the distances between the function addresses in each list, trying to find an address that, when added to each function address in the file, will produce an exact match with a corresponding address in the list from the crashed program. For example,

File addresses	Program addresses
12 (__junk)	A0502 (unknown1)
22 (__main)	A0530 (unknown2)
50 (__sub1)	A0570 (unknown3)
74 (__sub2)	A06E0 (unknown4)
90 (__sub3)	
140 (__sub4)	
200 (__sub5)	

If get_base can find a number that can be subtracted from each Program address and will yield an exact match with one of the File addresses, then that number is a plausible load address. Such a number for this example might be A04E0. This would make unknown1 match the name __main, unknown2 would match __sub1, unknown3 would match __sub3, and unknown4 would match __sub5.

To find the matches, get_base tries each possible base address in a loop, constructing the trial base address by subtracting each File address in turn from

the first Program address. In the loop, the routine `ismatch` is called to check all of the Program addresses for an exact match with the File addresses. If a match is found, that base address is stored in an array of base addresses called `bases`.

Because `get_trace` may not have had enough information to guarantee a correct list of Program addresses, it is possible that no exact matches were found. The `ismatch` function has maintained an index of the highest Program address that matches, just for this contingency. `Get_base` throws out this highest value, on the assumption that it was the value that caused the matches to fail. `Get_base` then calls itself to try again with the new (shorter) list, hoping for better luck now that the spurious entry is removed.

`get_base` returns the number of possible load addresses found and placed in the `bases` array.

Program 9-14. `getbase.c`

```
#include <debug.h>

int max_index          = 0;

get_base() {
    int x, y, bc;
    long int base;
    extern long int real_pcs[], bases[];
    extern int count, numpcs;
    extern struct sym syms[];

    max_index = 0;
    bc = 0;
    for( x = 0; x < count && numpcs > 0; x++ ) {
        base = real_pcs[0] - syms[x].value;
        if( ismatch( x, base, 1 ) ) {
            bases[bc++] = base;
        }
    }
    if( bc == 0 ) {
        if( max_index > 1 ) {
            for( x = max_index; x < numpcs; x++ )
                real_pcs[x] = real_pcs[x+1];
            numpcs--;
            return( get_base() );
        }
    }
    return(bc);
}
```

The `ismatch` Function

The first thing `ismatch`, Program 9-15, does is check to see if there are any Program addresses to match. If there are none, it returns 1 for success, because it has reached the end of the list without failing (all of which will be clearer in a moment). It then updates `max_index` for the benefit of `get_base`.

`ismatch` then loops through each File address, checking to see if the proposed base address, when added to the File address, matches the current Program address. If it does, then `ismatch` calls itself to process the rest of the

Program address list in an identical manner (this is how it can reach the end of the list and be successful). If the base address plus the File address are greater than the Program address, then there is no use looking any further, since the lists are sorted in increasing address order. `ismatch` returns 0 to report failure.

Notice that by calling itself recursively to process the remainder of the list, `ismatch` can deal with the problem of intervening subroutines in the File address list that are not in the Program address list (look at `_sub2` and `_sub4` in the example above). This problem is not a simple one to solve any other way, but `ismatch` does it simply and elegantly.

Program 9-15. `ismatch.c`

```
# include <debug.h>

ismatch( tab_index, base, pc_index )
int tab_index;
long int base;
int pc_index;{

    int x;
    extern long int real_pcs[];
    extern int count, max_index, numpcs;
    extern struct sym syms[];

    if( pc_index >= numpcs )
        return( 1 );
    if( pc_index > max_index ){
        max_index = pc_index;
    }
    for( x = 0; x < count; x++){
        if( syms[x].value + base == real_pcs[pc_index] ){
            return( ismatch( x, base, pc_index + 1 ) );
        }
        if( syms[x].value + base > real_pcs[pc_index] )
            return( 0 );
    }
    return(0);
}
```

The bt Function

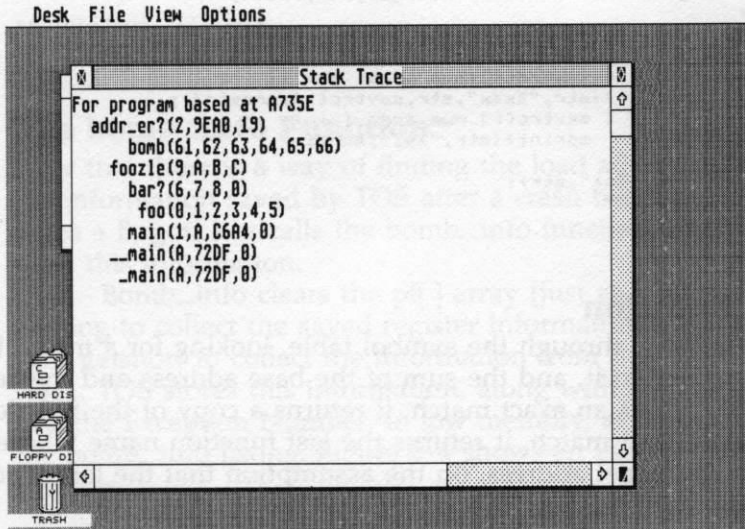
Now that all the possible base addresses have been found (there is usually only one, except in cases where the program crashed in the first or second subroutine in the program, and those problems are usually easy to solve), the `bt` function (program 9-16) can be called to print a stack trace for each possibility.

A stack trace looks like Figure 9-3. There is a line that shows the program load address (in this example there was a unique address found), and a list of lines that look like function calls, in the reverse order of when they were called (so that the user does not have to page down to see where the error occurred).

Note that some function names have a question mark after them. (See Figure 9-3.) This indicates that an exact address was not available, and a best guess was used. The guess is almost always correct, but can be wrong if the

crash damaged the stack in just the right way. The function names without question marks can always be trusted, since they were obtained by `get_realpc` from the code itself.

Figure 9-3. A Stack Trace



The `bt` function is simple, since all of the work has been done for it. It loops through the `savtrc` array built by `get_trace`, and calls `get_name` to find the name of the function in the symbol table, using the `real_pc` entry, and the base address supplied by `get_base`.

After printing the function name and an open parenthesis, it loops through all of the arguments collected by `get_args`, and prints them, followed by an end parenthesis. It leaves one blank line at the end of the list to separate it from any other traces that may follow (if there is more than one possible load address) and returns the next line that can be printed on. (The word “print” here is used loosely, since the data is really going into the `pl[]` array, which will be printed later by `showwnd`).

Program 9-16. `bt.c`

```
# include <debug.h>
# include <document.h>
# include <stdio.h>

bt(line,base)
int line;
long int base;{

    long int prog_base;
    char *str, buf[32], *get_name(), *exact_name();
    extern char pl[NLINES][NCHARS];
```

```

extern struct trc savtrc[];
int i, j;

sprintf(pl[line], "For program based at %X", base);
for( i = 0; savtrc[i].fp && i+line+1 < NLINES; i++ ){
    str = pl[i+line+1];
    if( savtrc[i].real_pc )
        get_name(savtrc[i].real_pc, buf, base);
    else
        get_name(savtrc[i].ret_pc, buf, base);
    sprintf(str, " %9.9s(", buf);
    for( j = 0; j < savtrc[i].num_args; j++ ){
        sprintf(str, "%s%x", str, savtrc[i].args[j]);
        if( j < savtrc[i].num_args-1 )
            sprintf(str, "%s, ", str);
    }
    sprintf(str, "%s)", str);
}
pl[i+line+1][0] = 0;
return(i+line+1);
}

```

The get_name Function

This function simply loops through the symbol table, looking for a match between the address passed to it, and the sum of the base address and the address in the table. If it finds an exact match, it returns a copy of the function name. If there was no exact match, it returns the last function name whose address was less than the target address, on the assumption that the target address must be inside that function. To indicate that this is merely an assumption (although almost always correct), it puts a question mark after the name. If there was no symbol table to look through, or the address was beyond all of the symbols, the address is returned in a string as a hexadecimal value.

Program 9-17. getname.c

```

#include <debug.h>

char *
get_name(addr, buf, base)
long int addr;
char *buf;
long int base;{

    int x;
    extern struct sym syms[];
    extern int count;

    if( count ){
        for( x = 1; x < count; x++ ){
            if( syms[x].value + base == addr ){
                if( syms[x].name[0] == '_' )
                    sprintf(buf, "%7.7s", &syms[x].name[1]);
                else
                    sprintf(buf, "%8.8s", syms[x].name);
                return(buf);
            }
            if( syms[x].value + base > addr ){
                if( x == 1 )
                    break;
                x--;
            }
        }
    }
}

```



```

        if( syms[x].name[0] == ' ' )
            sprintf(buf, "%7.7s?", &syms[x].name[1]);
        else
            sprintf(buf, "%8.8s?", syms[x].name);
        return(buf);
    }
}

sprintf(buf, "%X", addr);
return(buf);
}

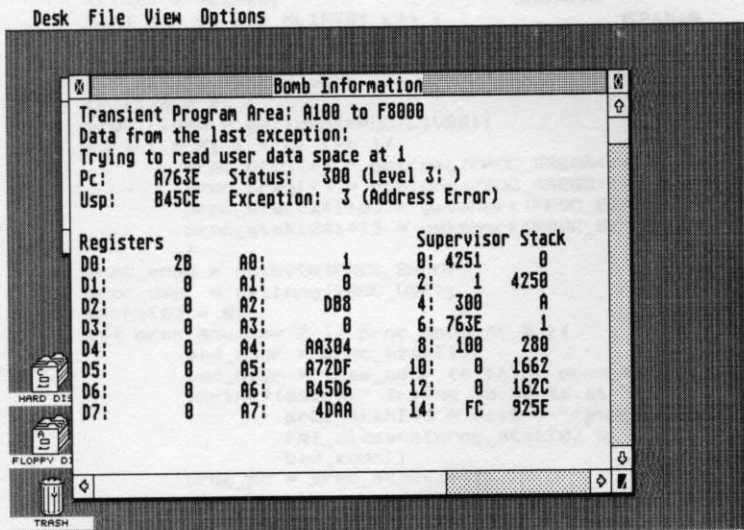
```

The bomb_info Function

Now that there is a way of finding the load address of the crashed program, the information saved by TOS after a crash becomes useful. When the user types a B, got_key calls the bomb_info function, Program 9-18, to format and print this information.

Bomb_info clears the pl[] array (just as trace did), and loops, calling getlong to collect the saved register information from the place TOS put it, and calls getshort to collect the information from the saved copy of the supervisor stack. TOS stores this information, along with the User Stack Pointer (USP) and the Exception Number, in low memory, at the addresses given at the top of the bomb_info listing. Figure 9-4 shows an example of a bomb information window.

Figure 9-4. A Bomb Information Window



The first thing that is printed in the window are the beginning and ending addresses of the Transient Program Area (TPA), the area of memory reserved for loading programs off of the disk. The crashed program must have

been loaded somewhere in this area, and any addresses that are outside of this area cannot belong to the program that crashed.

If the exception was a Bus Error or an Address Error, the extra information that was placed on the supervisor stack for these two exceptions is decoded and printed. This information tells what the program was attempting to do (read or write memory from the user or supervisor program or data space) and at what address it was trying to do it.

The program counter is located in the saved stack information in the same way that `get_trace` located it in the real stack earlier, by counting upwards past the status register and any saved data.

The program counter, status register, user stack pointer, and the exception number are printed, followed by the Registers and the saved supervisor stack information. The exception number is decoded, telling the user what it was that went wrong (in this case it was an Address Error).

Program 9-18. bombinfo.c

```
# include <document.h>
# define PROC_LIVES      0x380L
# define PROC_DREGS      0x384L
# define PROC_AREGS      0x3A4L
# define PROC_ENUM       0x3C4L
# define PROC_USP        0x3C8L
# define PROC_STACK      0x3CCL
# define STATMASK        0x58E0
# define MEMBOT          0x0432L
# define MEMTOP          0x0436L
# define TPASTART        0x0492L
# define TPALEN          0x0496L
```

```
char *Enums[] = {
    "Reset 0",
    "Reset 1",
    "Bus Error",
    "Address Error",
    "Illegal Instruction",
    "Zero Divide",
    "CHK instruction",
    "TRAPV instruction",
    "Privilege Violation",
    "Trace",
    "Line A",
    "Line F",
    "Unassigned 12",
    "Unassigned 13",
    "Format Error",
    "Uninitialized Interrupt",
    "Unassigned 16",
    "Unassigned 17",
    "Unassigned 18",
    "Unassigned 19",
    "Unassigned 20",
    "Unassigned 21",
    "Unassigned 22",
    "Unassigned 23",
    "Spurious Interrupt",
    "Level 1 Autovector",
    "Level 2 Autovector",
}
```

```

"Level 3 Autovector",
"Level 4 Autovector",
"Level 5 Autovector",
"Level 6 Autovector",
"Level 7 Autovector"
};
char $ref_classes[] = {
    "?",
    "user data space",
    "user program space",
    "??",
    "???",
    "supervisor data space",
    "supervisor program space",
    "interrupt acknowledge"
};
long int ref;
long int proc_enum, proc_esp, proc_pc;
long int proc_lives, proc_dreg[8], proc_areg[8];
int statusreg;
int tr_line;

bomb_info(whand,vw)
int whand, vw;{

    unsigned short int proc_stak[16];
    extern int main(), gl_hchar, gl_wchar, xlines;
    extern char $decode_status(), pl[NLINES][NCHARS];
    extern long int get_syms(), getlong();
    int i, x, y, w, h;
    int dx, dy, dw, dh, sx, sy, sw, sh;
    long int bad_addr;
    char extra[128];

    xlines = NLINES;
    for( x = 0; x < NLINES; x++ )
        pl[x][0] = 0;
    hide_mouse();
    clr_display(whand,vw);
    tr_line = 0;
    proc_lives = getlong(PROC_LIVES);
    for( i = 0; i < 8; i++ ){
        proc_dreg[i] = getlong(PROC_DREGS+i*sizeof(long));
        proc_areg[i] = getlong(PROC_AREGS+i*sizeof(long));
        proc_stak[2*i+0] = getshort(PROC_STACK+(2*i+0)*sizeof(short));
        proc_stak[2*i+1] = getshort(PROC_STACK+(2*i+1)*sizeof(short));
    }

    proc_enum = getbyte(PROC_ENUM);
    proc_esp = getlong(PROC_USP);
    extra[0] = 0;
    if( proc_enum == 2 || proc_enum == 3 ){
        bad_addr = proc_stak[1];
        bad_addr = (bad_addr << 16) | proc_stak[2];
        sprintf(extra, "Trying to %s %s at %X",
            proc_stak[0] & 0x10 ? "read" : "write",
            ref_classes[proc_stak[0] & 7],
            bad_addr);
        proc_pc = proc_stak[1+4];
        proc_pc = (proc_pc << 16) | proc_stak[2+4];
        statusreg = proc_stak[0+4];
    }

    else {
        proc_pc = proc_stak[1];
        proc_pc = (proc_pc << 16) | proc_stak[2];
        statusreg = proc_stak[0];
    }
}

```

```

sprintf(pl[tr_line++], " Transient Program Area: %X to %X",
        getlong(MEMBOT), getlong(MEMTOP) );
sprintf(pl[tr_line++], " Data from the last exception:");
sprintf(pl[tr_line++], extra);
sprintf(pl[tr_line++], " Pc: %8X   Status: %4x (%s)",
        proc_pc, statusreg, decode_status( statusreg ));
sprintf(pl[tr_line++], " Usp: %8X   Exception: %2X (%s)",
        proc_esp, proc_enum,
        proc_enum < 32 ? Enums[proc_enum & 0x1f] : "----");
sprintf(pl[tr_line++], "");
sprintf(pl[tr_line++],
        " Registers                                     Supervisor Stack");
for( i = 0; i < 8; i++ ){
    sprintf(pl[tr_line++],
            " D%d: %8X   A%d: %8X   %2.2d: %4x   %4x",
            i, proc_dreg[i], i, proc_areg[i], i*2,
            proc_stak[i*2], proc_stak[i*2+1]);
    }
show_mouse();
}

```

The decode_status Function

The decode status function, Program 9-19, decodes the status register, naming each of the seven important bits (if any are set) and printing the interrupt priority level at the time of the crash.

Program 9-19. decstat.c

```

char *
decode_status(stat)
int stat;{

    static char str[32];
    char *ptr;

    sprintf(str, "Level %d: ", (stat>>8)&7);
    ptr = &str[9];
    if( stat & 0x8000 )
        *ptr++ = 'T';
    if( stat & 0x2000 )
        *ptr++ = 'S';
    if( stat & 0x10 )
        *ptr++ = 'X';
    if( stat & 0x8 )
        *ptr++ = 'N';
    if( stat & 0x4 )
        *ptr++ = 'Z';
    if( stat & 0x2 )
        *ptr++ = 'V';
    if( stat & 0x1 )
        *ptr++ = 'C';
    *ptr = 0;
    return( str );
}

```

The do_new_window Function

After got_key has called trace or bombinfo, it calls do_new_window, Program 9-20, to create a window to hold the information. Do_new_window is almost exactly the same as the do_dir_window function in the shell, except that the

name and the `dis_window` variable have changed to make the routine more generic.

`Do_new_window` sets the variable `dis_window` to the new window handle so that `show_info` (called from `doit`) will know to call `showwnd`.

Program 9-20. `donewwnd.c`

```
# include <gemdefs.h>
# include <document.h>

# define BYE_BYE      -1
# define OBLIVION     -2

do_new_window(name,old_wh,old_vw,w,h)
char *name;
int old_wh, old_vw, w, h;{

    int whand, vp, hp, vs, hs, x, y, dx, dy, dw, dh, vw, events;
    int wlines, wcols, retval;
    extern int gl_wchar, gl_hchar, dis_window;
    extern int cur_line, cur_col, bold_line, highlight;

    w = gl_wchar;
    h = gl_hchar;
    wind_get( old_wh, WF_CURRXYWH, &x, &y, &dw, &dh );
    wind_get( 0, WF_WORKXYWH, &dx, &dy, &dw, &dh );
    if( w > dw-x )
        w = dw-x;
    if( h > dh-y )
        h = dh-y;

    wlines = h / gl_hchar;
    wcols = w / gl_wchar;
    cur_line = cur_col = 0;
    if( highlight && bold_line > wlines / 2 )
        cur_line = bold_line - wlines / 2;
    slide_pos( wlines, NLINES, cur_line, &vp );
    slide_pos( wcols, NCHARS, cur_col, &hp );
    slide_size( wlines, NLINES, &vs );
    slide_size( wcols, NCHARS, &hs );

    vw = old_vw;
    whand = new_window(name,1000-vp,hp,vs,hs,x,y,w,h,&vw);
    events = MU_MESAG | MU_KEYBD;
    dis_window = whand;
    retval = multi(events,&whand,0,name,&vw);
    close_window(whand);
    dis_window = 0;
    /*
    ** If the previous call to multi got an AC_CLOSE,
    ** then it returned OBLIVION. We must send another
    ** AC_CLOSE to the multi that is called by main(),
    ** so that the virtual workstation gets handled
    ** properly, and the other window gets closed properly.
    */
    if( retval == OBLIVION )
        close_me();
}

/*
** This routine sends a message to multi, faking an AC_CLOSE.
** This allows routines to be decoupled from actions that
** take place in was_msg(): the caller only needs to know that
** he wants to do whatever action AC_CLOSE causes, without having
```

CHAPTER 9

```
/* to know anything about the internal workings of was_msg.
*/
close_me(){
```

```
    int m[8];
    extern int gl_apid, menu_id, i_am_accessory;

    if( i_am_accessory ){
        m[0] = AC_CLOSE;
        m[3] = menu_id;
        m[1] = m[2] = m[4] = m[5] = m[6] = m[7] = 0;
        appl_write(gl_apid,16,m);
    }
}
```

The showwnd Function

The showwnd function, Program 9-21, sets up the slider positions in the now familiar manner, using `slide_pos` and `slide_size`, and then calls `just_clear` to clear the screen.

To actually print the information in the `pl[]` array onto the screen, `showwnd` loops, starting at `cur_line` and iterating up to the number of lines in the window. The special variables `highlight` and `bold_line` control a special feature used by the disassembly routines: One line is selected to be highlighted in boldface to indicate where in the disassembly the bomb occurred.

Program 9-21. showwnd.c

```
# include <gemdefs.h>
# include <document.h>

# define E_NORMAL          0
# define E_THICK           1
# define E_LIGHT           2
# define E_SKEWED          4
# define E_UNDERLINED      8
# define E_OUTLINED       16
# define E_SHADOWED       32

int highlight = 0;
int bold_line = 0;
char pl[NLINES][NCHARS];

showwnd(whand,vw)
int whand, vw;

{
    int x, y, w, h, i, hs, vs, hp, vp, count, wlines, wcols;
    extern int gl_wchar, gl_hchar, cur_col, cur_line;

    wind_get(whand,WF_WORKXYWH,&x,&y,&w,&h);
    count = cur_line + h / gl_hchar;
    if( count >= NLINES )
        count = NLINES - cur_line - 1;
    wlines = h / gl_hchar;
    wcols = w / gl_wchar;
    slide_pos( wlines, NLINES, cur_line, &vp );
    slide_pos( wcols, NCHARS, cur_col, &hp );
    slide_size( wlines, NLINES, &vs );
    slide_size( wcols, NCHARS, &hs );
    wind_set( whand, WF_VSLIDE, hp, 0, 0, 0 );
    wind_set( whand, WF_VSLIDE, vp, 0, 0, 0 );
}
```

```

wind_set( whand, WF_VSLSIZE, vs, 0, 0, 0 );
wind_set( whand, WF_HSLSIZE, hs, 0, 0, 0 );
just_clear(whand,vw);
hide_mouse();
for( i = cur_line; i < count; i++ )(
    if( strlen( plfil ) > cur_col )(
        if( highlight && bold_line == i )
            vst_effects( vw, E_THICK );
        v_gtext(vw, x, y+gl_hchar+(i-cur_line)*gl_hchar,
            &plfil[cur_col] );
        if( highlight && bold_line == i )
            vst_effects( vw, E_NORMAL );
    )
}
show_mouse();
}

```

The debug.h File

The file debug.h, Program 9-22, is an include file that contains macro definitions and structures used throughout this chapter.

Program 9-22.debug.h

```

# define NUM_ARGS      32

struct trc {
    short int *fp;
    short int *ret_pc;
    short int *real_pc;
    short int num_args;
    short int args[NUM_ARGS];
};

struct sym {
    char name[8];
    int type;
    long int value;
};

# ifdef HAS_DISASSEMBLY

# define BCDREG      0
# define BRANCH      1
# define CMPREG      2
# define EFFADD      3
# define EXAREG      4
# define EXDREG      5
# define IMMCCR      6
# define IMMSR       7
# define LINE_A      8
# define LINE_F      9
# define MOVEEA     10
# define MOVE_P     11
# define MOVE_Q     12
# define NONE       13
# define ONEREG     14
# define SFTROT     15
# define LINKOP     16
# define MOVEM      17
# define ADDR3A     18
# define IMMEA      19
# define MOVEAD     20
# define DBRNCH     21

```

CHAPTER 9

```
# define WEFFADD 22

struct hash_tab {
    char addrmode;
    char numhits;
    short value;
    short mask;
    char *string;
};

#
# define DATA_REG 0
# define ADDR_REG 1
# define INDIR 2
# define POSTINCR 3
# define PREDECR 4
# define DISPLACE 5
# define INDEXED 6
# define PCABSIMM 7
#
# define ABS_SHORT 0
# define ABS_LONG 1
# define PC_DISP 2
# define PC_INDEX 3
# define IMMEDIATE 4
#
# define BYTE 0
# define WORD 1
# define LONG 2

extern short int *Address, word, getshort();
extern char object[];

# define NEXTWORD() ( \
    word = getshort(Address++); \
    sprintf(object, "%s %04.4x", object, (unsigned short) word); \
)

# endif HAS_DISASSEMBLY
```

The linkone.bat and linkone.arg Files

To link the debugging aid accessory, the linkone.bat file and linkone.arg files are used.

The files that contain conditionally compiled code (such as the ones that reference the HAS_DISASSEMBLY macro) are separated in the argument file to remind programmers that they will need to be recompiled with the HAS_DISASSEMBLY macro defined if the disassembler is to be included.

The linkacc.bat and linkacc.arg files for the disassembler are shown in the next chapter.

Program 9-23. linkone.bat

```
c:\bin\link68 [undefined,symbols,command[linkone.arg]]
c:\bin\relmod a
c:\bin\rm a.68k
c:\bin\wait
```


Program 9-24. linkone.arg

```
a.68k=c:accstart.o,main.o,  
CONFIGAC.O,BT.O,DONEWWND.O,ISMATCH.O,GETBASE.O,DECSTAT.O,GETNAME.O,  
SHOWWND.O,GETSYMS.O,TRACE.O,OPENDATA.O,ERRORS.O,SETTOP.O,GETLONG.O,  
BOMBINFO.O,GETREAL.O,GETARGS.O,  
GIVEHELP.O,GOTKEY.O,DOIT.O,GETTRACE.O,  
accsup.o,env.a,vdibind,vdidata.o,gemlib,aesbind,osbind,libf
```

10 A Disassembler



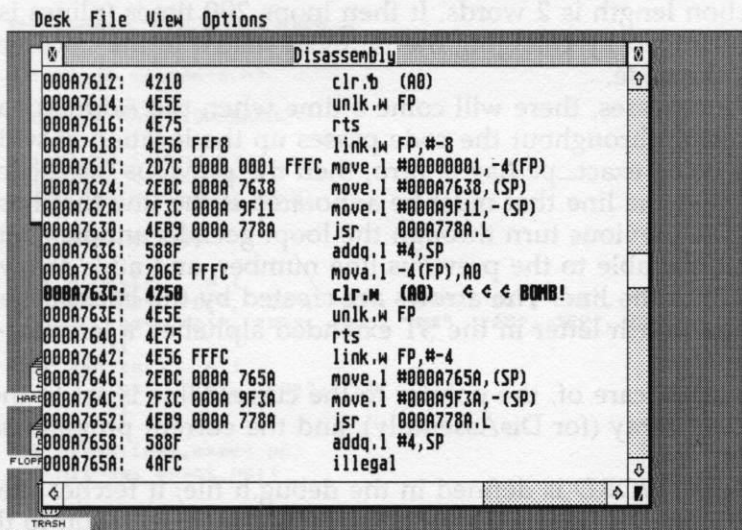
10 A Disassembler

10 A Disassembler

In this chapter, the debugging aid presented in Chapter 9 is augmented by a disassembler, a program that undoes the operation of an assembler, turning bits in the ST's memory back into symbolic assembly notation. This allows a programmer to understand the nature of the problem at the detailed level of the machine itself.

This tool is called an *annotated* disassembler because the instruction that caused the program to crash is printed out in boldface, and arrows point from the word BOMB! to the offending line. With this kind of exposure and the symbolic stack trace, program crashes become much easier to find and repair.

Figure 10-1. The disassembler prints out the instructions surrounding the point at which the program crashed.



At the time of a program crash, the desk accessory presented in the last chapter calls the `get_dis` routine presented here to capture the 100 instructions on either side of the program fault, and decodes them into humanly readable form before the operating system re-uses the dying program's memory. This decoded information can then be called up by the programmer for close examination after the operating system has cleaned up the mess.

Note that some programs can lock up the system in such a way as to preclude GEM from operating. This debugging aid is not designed to handle such problems, but the techniques shown in other parts of the book (notably the chapter on the Command Shell program) can be used to print the information gathered (stack trace, annotated disassembly) onto a disk file for perusal after the reset button is pressed. This is possible because most lock-ups affect only the higher-level GEM features such as mouse control of windows, and do not affect the low-level disk operations. *Just be sure to close the file so that all of the data is written to the disk before the reset.*

The `get_dis` Function

The `get_dis` routine, Program 10-1, is called by `get_trace` (described in the previous chapter), and is passed the number of lines to disassemble, and a flag that tells it whether the global variable `proc_pc` points to the instruction that failed, or to the instruction after the one that failed. (The 68000 processor detects some errors before advancing the program counter, but most are detected after the instruction is completed, or as the instruction's address is decoded, causing the need for the `exact_pc` flag.)

The `get_dis` function starts disassembling the code 200 words (a word is 16 bits) above the place where the program bombed, on the assumption that the average instruction length is 2 words. It then loops 200 times (`nlines` is 200, although it can be changed by changing the `NLINES` macro in the `document.h` file), disassembling the code.

As the loop progresses, there will come a time when the Address variable which is advancing throughout the code passes up the location at which the program failed. If the `exact_pc` flag is zero, then the previous instruction is the one that failed, and the line that must be annotated is the line that has been processed by the previous turn through the loop. `get_dis` annotates it by setting the `bold_line` variable to the previous line number, and adds arrows and the word BOMB! to the line. The arrows are created by the backslashed 4's, which denote the fourth letter in the ST extended alphabet, a leftward-pointing arrow.

Once that is taken care of, the pointer to the current line is set to the proper line in the `da[]` array (for DisAssembly), and the current program address is placed in the line.

The macro `NEXTWORD` is defined in the `debug.h` file; it fetches the next word out of the ST's memory and adds its hexadecimal representation to the character array object, which will be printed after the address when the instruction is fully decoded.

The disassembler consists of three parts. The first two are tables of decoded instructions called `table_A` and `table_B`. The third part is a set of subroutines that can decode the 68000's addressing modes. To get the instruction, the disassembler creates an index into `table_A` by putting the high ten bits of

the current word (fetched by NEXTWORD) into the variable hashA. The numhits attribute of each element of table_A tells whether the high ten bits of the instruction uniquely determine the instruction, or whether there was more than one instruction with the same ten high bits. If more than one instruction is indicated, table_B is searched for an exact match, via the matchB subroutine. If an exact match is found in table_B, then the table_B entry is used. If not, then the original table_A entry is used. This technique is probably familiar to you as a method of "hashing" (quickly selecting an item from a table by a calculation). The second table is known as an overflow table, to catch the items that the calculation did not uniquely address.

No matter which table the entry came from, the action performed is the same. The addrmode function is called to decode the addressing mode, and the hexadecimal object code (from the object array) is added to the line. If the instruction was a shift or a rotate instruction, it needs special treatment (because these instructions are not as regular in construction as the others), and it is built by combining the instruction from the table, an entry from the shifts array, and the addressing mode. If it was not a shift or rotate, then the instruction from the table and the decode addressing mode are combined. The result is put into the current line, and the line is annotated (if the exact_pc flag was nonzero, and the instruction just processed was the one that caused the error).

Program 10-1. getdis.c

```
#include <gemdefs.h>

# define HAS_DISASSEMBLY

#include <debug.h>
#include <document.h>
short int *Address, word;

char object[50];

char *regs[] = {
    "D0", "D1", "D2", "D3", "D4", "D5", "D6", "D7",
    "A0", "A1", "A2", "A3", "A4", "A5", "FP", "SP"
};
char *shifts[] = {
    "as", "ls", "rox", "ro"
};

get_dis(nlines, exact_pc)
int nlines, exact_pc; {

    static char buf[128], *str, *line;
    static int hashA, hashB, i;
    extern short int *proc_pc;
    short int *save_addr;
    extern char da[NLINES][NCHARS];
    extern char *addrmode();
    extern struct hash_tab table_A[], table_B[];
    extern short int getshort();
    extern long int getlong();
    extern int bold_line;
```



```

Address = &proc_pc[-nlines];
bold_line = -1;
line = buf;
buf[0] = 0;
for( i = 0; i < nlines; i++ ){
    save_addr = Address;
    if( exact_pc == 0 && bold_line < 0 && Address >= proc_pc ){
        bold_line = i-1;
        sprintf(line,"%s%s",line," \4 \4 \4 BOMB!");
    }
    line = buf;
    sprintf(line,"%08.BX: ",Address);
    object[0] = 0;
    NEXTWORD();
    hashA = (word >> 6) & 0x3ff;
    if( table_A[hashA].numhits && (hashB = matchB(word)) >= 0 ){
        str = addrmode(&table_B[hashB]);
        sprintf(line,"%s%-20.20s ",line,object);
        if( table_B[hashB].addrmode == SFTROT ){
            sprintf(buf,table_B[hashB].string,
                    shfts[(word>>3)&3],str);
        }
        else
            sprintf(buf,table_B[hashB].string,str,"oops!");
        sprintf(line,"%s%s",line,buf);
        if( exact_pc && bold_line < 0 && save_addr >= proc_pc ){
            bold_line = i;
            sprintf(line,"%s%s",line," \4 \4 BOMB!");
        }
        continue;
    }
    str = addrmode(&table_A[hashA]);
    sprintf(line,"%s%-20.20s ",line,object);
    if( table_A[hashA].addrmode == SFTROT ){
        sprintf(buf,table_A[hashA].string,
                shfts[(word>>3)&3],str);
    }
    else
        sprintf(buf,table_A[hashA].string,str,"OOPS!");
    sprintf(line,"%s%s",line,buf);
    if( exact_pc && bold_line < 0 && save_addr >= proc_pc ){
        bold_line = i;
        sprintf(line,"%s%s",line," \4 \4 BOMB!");
    }
}
}

```

The matchB Function

The matchB function, Program 10-2, is a simple linear search of the table_B array, masking the current instruction word and comparing it to the masked word from the array. The masking insures that only the bits that matter are compared, and the bits that have nothing to do with the instruction itself (such as the addressing mode bits) do not affect the comparison. Speed is not an issue here, or a faster search technique would be warranted. The table_B array is not often needed (most of the instructions are in the fast table_A array) and the most often used entries in the table_B array are placed at the front, where they will be found immediately. To compute the index, the address of the table is subtracted from the pointer that is pointing at the element found, and the result put in a long integer. (Even though the result is placed in a long integer, the

DRI C compiler warns that pointer subtraction yields long integers. Ignore this warning. Too bad the compiler is not smart enough to see that the code is using long integers, and eliminate the warning.)

Program 10-2. matchb.c

```
# define HAS_DISASSEMBLY

# include <debug.h>

matchB(wrd)
register short int wrd;{

    register struct hash_tab *ptr;
    extern struct hash_tab table_B[];
    long int retval;

    ptr = table_B;
    while( ptr->string ){
        if( (wrd & ptr->mask) == (ptr->value & ptr->mask) ){
            retval = ptr - table_B;
            return( retval );
        }
        ptr++;
    }
    return(-1);
}
```

The addrmode Function

The addrmode function, Program 10-3, is the longest routine in this book, but it is really just one large switch statement, whose regular structure makes it easy to understand despite its length.

Each element in the instruction decode tables contains an address mode field (tab->addrmode) which tells the addrmode function how to decode the address. The BRANCH mode means that the instruction was a branch of some sort, and addrmode decodes the branch address by adding the data after the instruction to the current address, and returning the result as a hexadecimal string in the buf array.

The DBRNCH mode is similar, but simpler, since the data is always one word long, instead of either a byte, a word, or a long word, as in the BRANCH case.

The ADDREA, MOVEAD, IMMEA, EFFADD, WEFFADD, MOVEM, and MOVEP modes all just set up a call to the effadd routine which will decode the effective address fields of the instruction. The pieces of the effective address field are the register, mode, and size, and are extracted from the instruction word and passed to effadd. The difference in each of these modes is mainly in the way the size field is calculated, and that is due to the somewhat haphazard way the size data was encoded when the 68000's instruction set was designed.

The MOVEEA mode is for the general MOVE instruction, which has two effective address fields, and can thus move data from memory to memory using a variety of addressing modes. Just to make things difficult, the size field is unlike that of the other instructions, and the register and mode fields in the

source operand are in the reverse order of those in the destination. The `effadd` routine is called twice to handle both operands.

The `LINKOP` mode is simple. The word after the instruction is printed in hexadecimal, taking care to put a minus sign in front of negative numbers. The `MOVE_Q` mode handles the move quick instruction's addressing mode, where the data is the low byte of the instruction itself.

The `SFTROT`, `BCDREG`, `COMPREG`, `EXAREG`, and `EXDREG` modes handle instructions that operate on registers or immediate data that is encoded in the instruction itself, just like the `MOVE_Q` mode.

Finally, for the `ONEREG`, `IMMCCR`, `IMMSR`, `LINE_A`, and `LINE_F` modes, nothing needs to be done, since the operands (if any) are encoded in the high ten bits of the instruction, and are thus handled by the hash tables directly. A question mark is returned, to make errors visible in case the code is being typed in or modified.

Program 10-3. `addrmode.c`

```
# define HAS_DISASSEMBLY

# include <debug.h>

char *
addrmode(tab)
struct hash_tab *tab;{

    static int mode, reg, size;
    static long int save, save2;
    static char byte;
    static short int op;
    static char buf[128];
    static char temp[128];
    extern char *effadd();
    extern short int getshort();
    extern long int getlong();
    extern char *regs[];

    switch(tab->addrmode){
        case BRANCH:
            op = word;
            save = (long int) Address;
            if( (op & 0xff) == 0 ){
                NEXTWORD();
                save += word;
            }
            else if( (op & 0xff) == 0xff ){
                NEXTWORD();
                save2 = word & 0xffff;
                NEXTWORD();
                save += (save2 << 16) | (word & 0xffff);
            }
            else {
                byte = word;
                save += byte;
            }
            sprintf(buf, "%08.8X", save);
            return( buf );
        case DBRNCH:
            save = (long int) Address;
            NEXTWORD();
```

```

    save += word;
    sprintf(buf, "%08.8X", save);
    return( buf );

case ADDREA:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = (word >> 8) & 1;
    if( size == 0 )
        size = WORD;
    else
        size = LONG;
    sprintf(buf, "%s", effadd( mode, reg, size ) );
    return( buf );

case MOVEAD:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = (word >> 12) & 3;
    if( size == 3 )
        size = WORD;
    else if( size == 2 )
        size = LONG;
    else
        return("Bad size in movea");
    sprintf(buf, "%s", effadd( mode, reg, size ) );
    return( buf );

case IMMEA:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = (word >> 6) & 3;
    NEXTWORD();
    save = (unsigned short int) word;
    if( size == LONG ) {
        NEXTWORD();
        save = (save << 16) | ((unsigned short) word);
        sprintf(buf, "%08.8X,%s",
            save, effadd( mode, reg, size ) );
    }
    else
        sprintf(buf, "%08.8X,%s",
            save, effadd( mode, reg, size ) );
    return(buf);

case EFFADD:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = (word >> 6) & 3;
    sprintf(buf, "%s", effadd( mode, reg, size ) );
    return( buf );

case WEFFADD:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = WORD;
    sprintf(buf, "%s", effadd( mode, reg, size ) );
    return( buf );

case LINKOP:
    NEXTWORD();
    if( word < 0 )
        sprintf(buf, "#-%x", -word);
    else
        sprintf(buf, "%x", word);
    return(buf);

case MOVEM:
    mode = (word >> 3) & 7;
    reg = word & 7;
    size = (word >> 6) & 3;
    NEXTWORD();
    save = word & 0xffff;

```

CHAPTER 10

```

        sprintf(buf, "%4.4X,%s",
            save, effadd( mode, reg, size ) );
        return(buf);
    case MOVEEA:
        mode = (word >> 3) & 7;
        reg = word & 7;
        size = (word >> 12) & 3;
        if( size == 1 )           /* size in the move instruction */
            size = BYTE; /* is different than in others */
        else if( size == 3 )
            size = WORD;
        else
            size = LONG;
        save = word;
        sprintf(temp, "%s", effadd( mode, reg, size ) );
        mode = (save >> 6) & 7;
        reg = (save >> 9) & 7;
        sprintf(buf, "%s,%s", temp, effadd( mode, reg, size ) );
        return( buf );
    case MOVE_Q:
        sprintf(buf, "%2.2x", word & 0xff);
        return( buf );
    case SFTROT:
        if( word & 0x20 )
            sprintf(buf, "D%d,D%d", (word>>9)&7, word&7);
        else
            sprintf(buf, "%d,D%d", (word>>9)&7, word&7);
        return( buf );
    case BCDREG:
        if( word & 8 )
            sprintf(buf, "-(%s),-(%s)",
                regs[8+word&7], regs[8+(word>>9)&7]);
        else
            sprintf(buf, "D%d,D%d", word&7, (word>>9)&7);
        return( buf );
    case CMPREG:
        sprintf(buf, "(%s)+, (%s)+",
            regs[8+word&7], regs[8+(word>>9)&7]);
        return( buf );
    case MOVE_P:
        size = word & 0x40;
        if( size )
            size = LONG;
        else
            size = WORD;
        sprintf(buf, "%s", effadd( 5, word & 7, size ) );
        return(buf);
    case EXAREG:
        sprintf(buf, "%s", regs[8+word&7]);
        return(buf);
    case EXDREG:
        sprintf(buf, "D%d", word&7);
        return(buf);
    case ONEREG:
    case IMMCCR:
    case IMMSR:
    case LINE_A:
    case LINE_F:
    case NONE:
    default:
        return("?");
}

```


The effadd Function

The `effadd` function, Program 10-4, handles the simple cases directly, and hands the complicated case to the `pcabsimm` routine. The simplest case is the *data register direct* mode, where all that is required is the name of the data register. Since `addrmode` passed the register number to `effadd`, all `effadd` needs to do is print it with a D in front.

The *address register direct* mode is similar, but an array of address register names is used so that the alternate forms FP and SP can be used for the registers A6 and A7, to make their use as the Frame Pointer and Stack Pointer apparent.

The `INDIR`, `POSTINCR`, and `PREDECR` modes all enclose an address register inside parentheses to indicate indirection, with pluses and minuses to indicate incrementing and decrementing. The `DISPLACE` mode is similar, except that the next word is fetched and printed as an offset before the register.

The `INDEXED` mode fetches the index word, checks it for validity, and decodes it, putting the displacement, index register, address register, and size in their proper location.

Last, the `PCABSIMM` mode is handled by calling the `pcabsimm` routine.

Program 10-4. `effadd.c`

```
# define HAS_DISASSEMBLY

# include <debug.h>

char *
effadd(mode, reg, size)
int mode, reg, size; {

    static char string[128], byte;
    extern short int getshort();
    extern long int getlong();
    extern char *regs[];
    extern char *pcabsimm();

    switch(mode) {
    case DATA_REG:
        sprintf(string, "D%o", reg);
        break;
    case ADDR_REG:
        sprintf(string, "%s", regs[reg+8]);
        break;
    case INDIR:
        sprintf(string, "(%s)", regs[reg+8]);
        break;
    case POSTINCR:
        sprintf(string, "(%s)+", regs[reg+8]);
        break;
    case PREDECR:
        sprintf(string, "-(%s)", regs[reg+8]);
        break;
    case DISPLACE:
        NEXTWORD();
        sprintf(string, "%d(%s)", word, regs[reg+8]);
        break;
    case INDEXED:
```

```

NEXTWORD();
if( word & 0x0700 ) (
    sprintf(string, "Bad indexed addressing mode");
}
else (
    byte = word;
    sprintf(string, "%d(%s,%s.%c)",
        byte, regs[reg+8], regs[(word>>12)&0xf],
        (word & 0x0800) ? 'l' : 'w' );
)
break;
case PCABSIMM:
    pcabsimm(string, mode, reg, size);
    break;
}
return(string);
}

```

The pcabsimm Function

The pcabsimm function handles the absolute addressing modes, and the program counter relative modes directly, and in the tradition established in the effadd routine, handles the hard part by calling the immediate routine to handle the immediate modes.

In the ABS_SHORT and ABS_LONG modes, the address is the short or long word following the instruction. Pcabsimm converts the address to hexadecimal and returns the result string. The PC_DISP and PC_INDEX are very much like the DISPLACE and INDEXED modes in effadd, except that since the address is relative to the current program counter, which is known at this point (it is pointing to this instruction), the true target address can be printed in square brackets after the decoded addressing mode. As mentioned, the immediate modes are handled by calling immediate.

Program 10-5. pcabsimm.c

```

#define HAS_DISASSEMBLY

#include <debug.h>

char *
pcabsimm(string, mode, reg, size)
char *string;
int mode, reg, size; {

    static long longword, offset;
    static char byte;
    extern short int getshort();
    extern long int getlong();
    extern char *regs[];
    extern char *immediate();

    switch(reg) {
        case ABS_SHORT:
            NEXTWORD();
            sprintf(string, "%04.4x.W", (unsigned short int) word);
            return(string);
        case ABS_LONG:
            NEXTWORD();
            longword = word;

```

```

NEXTWORD();
sprintf(string, "%08.8X.L",
        (longword << 16) | ((unsigned short int) word));
return(string);
case PC_DISP:
    offset = (long int) Address;
    NEXTWORD();
    offset += word;
    sprintf(string, "%04.4x(pc) [%X]",
            (unsigned short int) word, offset);
    return(string);
case PC_INDEX:
    offset = (long int) Address;
    NEXTWORD();
    byte = word;
    offset += byte;
    sprintf(string, "%.2x(pc, %s.%c) [%X]",
            byte, regs[(word >> 12) & 0xf],
            (word & 0x0800) ? '1' : 'w', offset);
    return(string);
case IMMEDIATE:
    immediate(string, size);
    return(string);
default:
    return("??");
}

```

The immediate Function

The immediate function handles byte-, word-, and long word-sized immediate data.

The BYTE mode reads the next word, and uses only the low byte of it. The WORD mode uses the entire word, and the LONG mode fetches two words, and patches them together into a long word. All modes convert the results to hexadecimal and return.

Program 10-6. immediat.c

```

#define HAS_DISASSEMBLY

#include <debug.h>

char *
immediate(string, size)
char *string;
int size;
{
    static long int longword;
    extern short int getshort();
    extern long int getlong();

    switch(size) {
        case BYTE:
            NEXTWORD();
            sprintf(string, "%04.4x", word & 0xff);
            return(string);
        default:
            case WORD:
                NEXTWORD();
                sprintf(string, "%04.4x", (unsigned short int) word);
                return(string);
    }
}

```

```

case LONG:
    NEXTWORD();
    longword = word;
    NEXTWORD();
    sprintf(string, "%X08.0X",
        (longword << 16) | ((unsigned short int) word));
    return(string);
}

```

The disassem Function

Once the `get_dis` function has captured the information it needs, it returns to `get_trace`, which eventually returns to the exception handler routine, which jumps to TOS to print the bomb icons and clean up. To have the desk accessory show the disassembly listing, the user presses the D key, and the `disassem` function, Program 10-7, is called.

The `disassem` function calls `setup_dis` to set the slider values (since the disassembly is several pages long, the slider will be needed to allow the user to display the parts that don't fit on the initial screen). Then `disassem` loops through all the lines, copying the `da[]` array which holds the disassembly listing into the `pl[]` array, which is what `showwnd` prints on the screen.

After the `disassem` function has been called (by `got_key`), `got_key` calls the `do_new_wnd` routine to create a window into which `showwnd` will print the `pl[]` array. GEM will send multi a REDRAW message, which will be passed to `wasmgs`, which calls `do_redraw`, which calls `just_draw`, which calls `doit`, which calls `show_info`, which calls `showwnd`, which prints the `pl[]` array.

Program 10-7. `disassem.c`

```

#include <document.h>

char da[NLINES][NCHARS];

disassem(whand,vw,nlines)
int whand, vw, nlines;{

    int i;
    extern int xlines;
    extern char pl[NLINES][NCHARS];

    xlines = nlines;
    setup_dis(whand,vw,nlines);
    for( i = 0; i < NLINES; i++ )
        strncpy( pl[i], da[i], NCHARS );
}

```

The setup_dis Function

The `setup_dis` function, Program 10-8, should look familiar by now. It calls `wind_get`, `slide_size`, `slide_pos`, and `wind_set` to set the sliders to the proper size and position to reflect the size and position of the window within the entire disassembly listing.

Program 10-8. setupdis.c

```
# include <gemdefs.h>

setup_dis(whand,vw,nlines)
int whand, vw, nlines;{

    static int x, y, w, h, wlines, hs, vs, hp, vp;
    extern int gl_hchar;

    wind_get( whand, WF_WORKXYWH, &x, &y, &w, &h );
    wlines = h / gl_hchar;
    slide_size( wlines, nlines, &vs );
    slide_size( 1, 1, &hs );
    slide_pos( wlines, nlines, 0, &vp );
    slide_pos( wlines, nlines, 0, &hp );
    wind_set( whand, WF_VSLSIZE, vs, 0, 0, 0 );
    wind_set( whand, WF_HSLSIZE, hs, 0, 0, 0 );
    wind_set( whand, WF_VSLIDE, vp, 0, 0, 0 );
    wind_set( whand, WF_HSLIDE, hp, 0, 0, 0 );
}
```

The tables.c File

The tables table_A and table_B are long (Program 10-9), but their format is regular enough that typing them in with a good editor does not really take that long.

The hash_tab structure is defined in debug.h, and has five elements. These are the addressing mode (addrmode), the number of instructions that share the same high ten bits as this one (numhits), the instruction itself (value), a mask to tell which bits in the instruction are relevant to the instruction and which are irrelevant (for instance, which are parts of the addressing mode and not parts of the instruction), and a string that has the decode name of the instruction, usually with a format field (%s) which sprintf will use to place the decoded address in the right place in the string.

Program 10-9. tables.c

```
#

# define HAS_DISASSEMBLY

# include <debug.h>

char unimplemented[] = "unimplemented";

struct hash_tab table_A[] = {
/* A: 0x0000 */ IMMEA, 0, 0x0000, 0xffc0, "ori.b %s",
/* A: 0x0040 */ IMMEA, 0, 0x0040, 0xffc0, "ori.w %s",
/* A: 0x0080 */ IMMEA, 0, 0x0080, 0xffc0, "ori.l %s",
/* A: 0x00c0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x0100 */ EFFADD, 1, 0x0100, 0xffc0, "btst.l D0,%s",
/* A: 0x0140 */ EFFADD, 1, 0x0140, 0xffc0, "bchg.l D0,%s",
/* A: 0x0180 */ EFFADD, 1, 0x0180, 0xffc0, "bclr.l D0,%s",
/* A: 0x01c0 */ EFFADD, 1, 0x01c0, 0xffc0, "bset.l D0,%s",
/* A: 0x0200 */ IMMEA, 1, 0x0200, 0xffc0, "andi.b %s",
/* A: 0x0240 */ IMMEA, 1, 0x0240, 0xffc0, "andi.w %s",
/* A: 0x0280 */ IMMEA, 0, 0x0280, 0xffc0, "andi.l %s",
/* A: 0x02c0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
```

CHAPTER 10

```

/* A: 0x0300 */ EFFADD, 1, 0x0300, 0xffc0, "btst.1 D1,%s",
/* A: 0x0340 */ EFFADD, 1, 0x0340, 0xffc0, "bchg.1 D1,%s",
/* A: 0x0380 */ EFFADD, 1, 0x0380, 0xffc0, "bclr.1 D1,%s",
/* A: 0x03c0 */ EFFADD, 1, 0x03c0, 0xffc0, "bset.1 D1,%s",
/* A: 0x0400 */ IMMEA, 0, 0x0400, 0xffc0, "subi.b %s",
/* A: 0x0440 */ IMMEA, 0, 0x0440, 0xffc0, "subi.w %s",
/* A: 0x0480 */ IMMEA, 0, 0x0480, 0xffc0, "subi.l %s",
/* A: 0x04c0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0500 */ EFFADD, 1, 0x0500, 0xffc0, "btst.1 D2,%s",
/* A: 0x0540 */ EFFADD, 1, 0x0540, 0xffc0, "bchg.1 D2,%s",
/* A: 0x0580 */ EFFADD, 1, 0x0580, 0xffc0, "bclr.1 D2,%s",
/* A: 0x05c0 */ EFFADD, 1, 0x05c0, 0xffc0, "bset.1 D2,%s",
/* A: 0x0600 */ IMMEA, 0, 0x0600, 0xffc0, "addi.b %s",
/* A: 0x0640 */ IMMEA, 0, 0x0640, 0xffc0, "addi.w %s",
/* A: 0x0680 */ IMMEA, 0, 0x0680, 0xffc0, "addi.l %s",
/* A: 0x06c0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0700 */ EFFADD, 1, 0x0700, 0xffc0, "btst.1 D3,%s",
/* A: 0x0740 */ EFFADD, 1, 0x0740, 0xffc0, "bchg.1 D3,%s",
/* A: 0x0780 */ EFFADD, 1, 0x0780, 0xffc0, "bclr.1 D3,%s",
/* A: 0x07c0 */ EFFADD, 1, 0x07c0, 0xffc0, "bset.1 D3,%s",
/* A: 0x0800 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0840 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0880 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x08c0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0900 */ EFFADD, 1, 0x0900, 0xffc0, "btst.1 D4,%s",
/* A: 0x0940 */ EFFADD, 1, 0x0940, 0xffc0, "bchg.1 D4,%s",
/* A: 0x0980 */ EFFADD, 1, 0x0980, 0xffc0, "bclr.1 D4,%s",
/* A: 0x09c0 */ EFFADD, 1, 0x09c0, 0xffc0, "bset.1 D4,%s",
/* A: 0x0a00 */ IMMEA, 1, 0x0a00, 0xffc0, "eori.b %s",
/* A: 0x0a40 */ IMMEA, 1, 0x0a40, 0xffc0, "eori.w %s",
/* A: 0x0a80 */ IMMEA, 0, 0x0a80, 0xffc0, "eori.l %s",
/* A: 0x0ac0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0b00 */ EFFADD, 1, 0x0b00, 0xffc0, "btst.1 D5,%s",
/* A: 0x0b40 */ EFFADD, 1, 0x0b40, 0xffc0, "bchg.1 D5,%s",
/* A: 0x0b80 */ EFFADD, 1, 0x0b80, 0xffc0, "bclr.1 D5,%s",
/* A: 0x0bc0 */ EFFADD, 1, 0x0bc0, 0xffc0, "bset.1 D5,%s",
/* A: 0x0c00 */ IMMEA, 0, 0x0c00, 0xffc0, "cmpi.b %s",
/* A: 0x0c40 */ IMMEA, 0, 0x0c40, 0xffc0, "cmpi.w %s",
/* A: 0x0c80 */ IMMEA, 0, 0x0c80, 0xffc0, "cmpi.l %s",
/* A: 0x0cc0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0d00 */ EFFADD, 1, 0x0d00, 0xffc0, "btst.1 D6,%s",
/* A: 0x0d40 */ EFFADD, 1, 0x0d40, 0xffc0, "bchg.1 D6,%s",
/* A: 0x0d80 */ EFFADD, 1, 0x0d80, 0xffc0, "bclr.1 D6,%s",
/* A: 0x0dc0 */ EFFADD, 1, 0x0dc0, 0xffc0, "bset.1 D6,%s",
/* A: 0x0e00 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0e40 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0e80 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0ec0 */ NONE, 0, 0x0000, 0x0000, "unimplemented",
/* A: 0x0f00 */ EFFADD, 1, 0x0f00, 0xffc0, "btst.1 D7,%s",
/* A: 0x0f40 */ EFFADD, 1, 0x0f40, 0xffc0, "bchg.1 D7,%s",
/* A: 0x0f80 */ EFFADD, 1, 0x0f80, 0xffc0, "bclr.1 D7,%s",
/* A: 0x0fc0 */ EFFADD, 1, 0x0fc0, 0xffc0, "bset.1 D7,%s",
/* A: 0x1000 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1040 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1080 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x10c0 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1100 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1140 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1180 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x11c0 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1200 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1240 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1280 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x12c0 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1300 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1340 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",
/* A: 0x1380 */ MOVEEA, 0, 0x1000, 0xf000, "move.b %s",

```


[illegible]


```

/* A: 0x3540 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3580 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x35c0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3600 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3640 */ MOVEAD, 0, 0x3640, 0xffc0, "move.w %s,A3",
/* A: 0x3680 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x36c0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3700 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3740 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3780 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x37c0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3800 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3840 */ MOVEAD, 0, 0x3840, 0xffc0, "move.w %s,A4",
/* A: 0x3880 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x38c0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3900 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3940 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3980 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x39c0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3a00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3a40 */ MOVEAD, 0, 0x3a40, 0xffc0, "move.w %s,A5",
/* A: 0x3a80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3ac0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3b00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3b40 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3b80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3bc0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3c00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3c40 */ MOVEAD, 0, 0x3c40, 0xffc0, "move.w %s,FP",
/* A: 0x3c80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3cc0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3d00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3d40 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3d80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3dc0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3e00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3e40 */ MOVEAD, 0, 0x3e40, 0xffc0, "move.w %s,SP",
/* A: 0x3e80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3ec0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3f00 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3f40 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3f80 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x3fc0 */ MOVEEA, 0, 0x3000, 0xf000, "move.w %s",
/* A: 0x4000 */ EFFADD, 0, 0x4000, 0xffc0, "neg.b %s",
/* A: 0x4040 */ EFFADD, 0, 0x4060, 0xffc0, "move.l SR,%s",
/* A: 0x4080 */ EFFADD, 0, 0x4080, 0xffc0, "neg.l %s",
/* A: 0x40c0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4100 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4140 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4180 */ WEFFADD, 0, 0x4180, 0xffc0, "chk.w D0,%s",
/* A: 0x41c0 */ EFFADD, 0, 0x41c0, 0xffc0, "lea.l %s,A0",
/* A: 0x4200 */ EFFADD, 0, 0x4200, 0xffc0, "clr.b %s",
/* A: 0x4240 */ EFFADD, 0, 0x4240, 0xffc0, "clr.w %s",
/* A: 0x4280 */ EFFADD, 0, 0x4280, 0xffc0, "clr.l %s",
/* A: 0x42c0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4300 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4340 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4380 */ WEFFADD, 0, 0x4380, 0xffc0, "chk.w D1,%s",
/* A: 0x43c0 */ EFFADD, 0, 0x43c0, 0xffc0, "lea.l %s,A1",
/* A: 0x4400 */ EFFADD, 0, 0x4400, 0xffc0, "neg.b %s",
/* A: 0x4440 */ EFFADD, 0, 0x4440, 0xffc0, "neg.w %s",
/* A: 0x4480 */ EFFADD, 0, 0x4480, 0xffc0, "neg.l %s",
/* A: 0x44c0 */ EFFADD, 0, 0x44c0, 0xffc0, "move %s,CCR",
/* A: 0x4500 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4540 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4580 */ WEFFADD, 0, 0x4580, 0xffc0, "chk.w D2,%s",
/* A: 0x45c0 */ EFFADD, 0, 0x45c0, 0xffc0, "lea.l %s,A2",

```

CHAPTER 10

```

/* A: 0x4600 */ EFFADD, 0, 0x4600, 0xffc0, "not.b %s",
/* A: 0x4640 */ EFFADD, 0, 0x4640, 0xffc0, "not.w %s",
/* A: 0x4680 */ EFFADD, 0, 0x4680, 0xffc0, "not.l %s",
/* A: 0x46c0 */ EFFADD, 0, 0x46c0, 0xffc0, "move.l %s,SR",
/* A: 0x4700 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4740 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4780 */ WEFFADD, 0, 0x4780, 0xffc0, "chk.w D3,%s",
/* A: 0x47c0 */ EFFADD, 0, 0x47c0, 0xffc0, "lea.l %s,A3",
/* A: 0x4800 */ EFFADD, 0, 0x4800, 0xffc0, "nbcd %s",
/* A: 0x4840 */ EFFADD, 8, 0x4840, 0xffc0, "pea.l %s",
/* A: 0x4880 */ MOVEM, 8, 0x4880, 0xffc0, "movm.w %s",
/* A: 0x48c0 */ MOVEM, 8, 0x48c0, 0xffc0, "movm.l %s",
/* A: 0x4900 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4940 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4980 */ WEFFADD, 0, 0x4980, 0xffc0, "chk.w D4,%s",
/* A: 0x49c0 */ EFFADD, 0, 0x49c0, 0xffc0, "lea.l %s,A4",
/* A: 0x4a00 */ EFFADD, 0, 0x4a00, 0xffc0, "tst.b %s",
/* A: 0x4a40 */ EFFADD, 0, 0x4a40, 0xffc0, "tst.w %s",
/* A: 0x4a80 */ EFFADD, 0, 0x4a80, 0xffc0, "tst.l %s",
/* A: 0x4ac0 */ EFFADD, 1, 0x4ac0, 0xffc0, "tas.b %s",
/* A: 0x4b00 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4b40 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4b80 */ WEFFADD, 0, 0x4b80, 0xffc0, "chk.w D5,%s",
/* A: 0x4bc0 */ EFFADD, 0, 0x4bc0, 0xffc0, "lea.l %s,A5",
/* A: 0x4c00 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4c40 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4c80 */ MOVEM, 0, 0x4c80, 0xffc0, "movm.w %s",
/* A: 0x4cc0 */ MOVEM, 0, 0x4cc0, 0xffc0, "movm.l %s",
/* A: 0x4d00 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4d40 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4d80 */ WEFFADD, 0, 0x4d80, 0xffc0, "chk.w D6,%s",
/* A: 0x4dc0 */ EFFADD, 0, 0x4dc0, 0xffc0, "lea.l %s,FP",
/* A: 0x4e00 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4e40 */ ONEREG, 54, 0x4e40, 0xffff, "trap #0",
/* A: 0x4e80 */ EFFADD, 0, 0x4e80, 0xffc0, "jsr %s",
/* A: 0x4ec0 */ EFFADD, 0, 0x4ec0, 0xffc0, "jmp %s",
/* A: 0x4f00 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4f40 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0x4f80 */ WEFFADD, 0, 0x4f80, 0xffc0, "chk.w D7,%s",
/* A: 0x4fc0 */ EFFADD, 0, 0x4fc0, 0xffc0, "lea.l %s,SP",
/* A: 0x5000 */ EFFADD, 0, 0x5000, 0xffc0, "addq.l #8,%s",
/* A: 0x5040 */ EFFADD, 0, 0x5040, 0xffc0, "addq.l #8,%s",
/* A: 0x5080 */ EFFADD, 0, 0x5080, 0xffc0, "addq.l #8,%s",
/* A: 0x50c0 */ EFFADD, 8, 0x50c0, 0xffc0, "st.b %s",
/* A: 0x5100 */ EFFADD, 0, 0x5100, 0xffc0, "subq.l #8,%s",
/* A: 0x5140 */ EFFADD, 0, 0x5140, 0xffc0, "subq.l #8,%s",
/* A: 0x5180 */ EFFADD, 0, 0x5180, 0xffc0, "subq.l #8,%s",
/* A: 0x51c0 */ EFFADD, 8, 0x51c0, 0xffc0, "sf.b %s",
/* A: 0x5200 */ EFFADD, 0, 0x5200, 0xffc0, "addq.l #1,%s",
/* A: 0x5240 */ EFFADD, 0, 0x5240, 0xffc0, "addq.l #1,%s",
/* A: 0x5280 */ EFFADD, 0, 0x5280, 0xffc0, "addq.l #1,%s",
/* A: 0x52c0 */ EFFADD, 8, 0x52c0, 0xffc0, "shi.b %s",
/* A: 0x5300 */ EFFADD, 0, 0x5300, 0xffc0, "subq.l #1,%s",
/* A: 0x5340 */ EFFADD, 0, 0x5340, 0xffc0, "subq.l #1,%s",
/* A: 0x5380 */ EFFADD, 0, 0x5380, 0xffc0, "subq.l #1,%s",
/* A: 0x53c0 */ EFFADD, 8, 0x53c0, 0xffc0, "sls.b %s",
/* A: 0x5400 */ EFFADD, 0, 0x5400, 0xffc0, "addq.l #2,%s",
/* A: 0x5440 */ EFFADD, 0, 0x5440, 0xffc0, "addq.l #2,%s",
/* A: 0x5480 */ EFFADD, 0, 0x5480, 0xffc0, "addq.l #2,%s",
/* A: 0x54c0 */ EFFADD, 8, 0x54c0, 0xffc0, "scc.b %s",
/* A: 0x5500 */ EFFADD, 0, 0x5500, 0xffc0, "subq.l #2,%s",
/* A: 0x5540 */ EFFADD, 0, 0x5540, 0xffc0, "subq.l #2,%s",
/* A: 0x5580 */ EFFADD, 0, 0x5580, 0xffc0, "subq.l #2,%s",
/* A: 0x55c0 */ EFFADD, 8, 0x55c0, 0xffc0, "scs.b %s",
/* A: 0x5600 */ EFFADD, 0, 0x5600, 0xffc0, "addq.l #3,%s",
/* A: 0x5640 */ EFFADD, 0, 0x5640, 0xffc0, "addq.l #3,%s",
/* A: 0x5680 */ EFFADD, 0, 0x5680, 0xffc0, "addq.l #3,%s",

```

```

/* A: 0x56c0 */ EFFADD, 8, 0x56c0, 0xffc0, "sne.b %s",
/* A: 0x5700 */ EFFADD, 0, 0x5700, 0xffc0, "subq.l #3,%s",
/* A: 0x5740 */ EFFADD, 0, 0x5740, 0xffc0, "subq.l #3,%s",
/* A: 0x5780 */ EFFADD, 0, 0x5780, 0xffc0, "subq.l #3,%s",
/* A: 0x57c0 */ EFFADD, 8, 0x57c0, 0xffc0, "seq.b %s",
/* A: 0x5800 */ EFFADD, 0, 0x5800, 0xffc0, "addq.l #4,%s",
/* A: 0x5840 */ EFFADD, 0, 0x5840, 0xffc0, "addq.l #4,%s",
/* A: 0x5880 */ EFFADD, 0, 0x5880, 0xffc0, "addq.l #4,%s",
/* A: 0x58c0 */ EFFADD, 8, 0x58c0, 0xffc0, "svc.b %s",
/* A: 0x5900 */ EFFADD, 0, 0x5900, 0xffc0, "subq.l #4,%s",
/* A: 0x5940 */ EFFADD, 0, 0x5940, 0xffc0, "subq.l #4,%s",
/* A: 0x5980 */ EFFADD, 0, 0x5980, 0xffc0, "subq.l #4,%s",
/* A: 0x59c0 */ EFFADD, 8, 0x59c0, 0xffc0, "svs.b %s",
/* A: 0x5a00 */ EFFADD, 0, 0x5a00, 0xffc0, "addq.l #5,%s",
/* A: 0x5a40 */ EFFADD, 0, 0x5a40, 0xffc0, "addq.l #5,%s",
/* A: 0x5a80 */ EFFADD, 0, 0x5a80, 0xffc0, "addq.l #5,%s",
/* A: 0x5ac0 */ EFFADD, 8, 0x5ac0, 0xffc0, "spl.b %s",
/* A: 0x5b00 */ EFFADD, 0, 0x5b00, 0xffc0, "subq.l #5,%s",
/* A: 0x5b40 */ EFFADD, 0, 0x5b40, 0xffc0, "subq.l #5,%s",
/* A: 0x5b80 */ EFFADD, 0, 0x5b80, 0xffc0, "subq.l #5,%s",
/* A: 0x5bc0 */ EFFADD, 8, 0x5bc0, 0xffc0, "smi.b %s",
/* A: 0x5c00 */ EFFADD, 0, 0x5c00, 0xffc0, "addq.l #6,%s",
/* A: 0x5c40 */ EFFADD, 0, 0x5c40, 0xffc0, "addq.l #6,%s",
/* A: 0x5c80 */ EFFADD, 0, 0x5c80, 0xffc0, "addq.l #6,%s",
/* A: 0x5cc0 */ EFFADD, 8, 0x5cc0, 0xffc0, "sge.b %s",
/* A: 0x5d00 */ EFFADD, 0, 0x5d00, 0xffc0, "subq.l #6,%s",
/* A: 0x5d40 */ EFFADD, 0, 0x5d40, 0xffc0, "subq.l #6,%s",
/* A: 0x5d80 */ EFFADD, 0, 0x5d80, 0xffc0, "subq.l #6,%s",
/* A: 0x5dc0 */ EFFADD, 8, 0x5dc0, 0xffc0, "slt.b %s",
/* A: 0x5e00 */ EFFADD, 0, 0x5e00, 0xffc0, "addq.l #7,%s",
/* A: 0x5e40 */ EFFADD, 0, 0x5e40, 0xffc0, "addq.l #7,%s",
/* A: 0x5e80 */ EFFADD, 0, 0x5e80, 0xffc0, "addq.l #7,%s",
/* A: 0x5ec0 */ EFFADD, 8, 0x5ec0, 0xffc0, "sgt.b %s",
/* A: 0x5f00 */ EFFADD, 0, 0x5f00, 0xffc0, "subq.l #7,%s",
/* A: 0x5f40 */ EFFADD, 0, 0x5f40, 0xffc0, "subq.l #7,%s",
/* A: 0x5f80 */ EFFADD, 0, 0x5f80, 0xffc0, "subq.l #7,%s",
/* A: 0x5fc0 */ EFFADD, 8, 0x5fc0, 0xffc0, "sle.b %s",
/* A: 0x6000 */ BRANCH, 0, 0x6000, 0xff00, "bra.b %s",
/* A: 0x6040 */ BRANCH, 0, 0x6040, 0xff00, "bra.b %s",
/* A: 0x6080 */ BRANCH, 0, 0x6080, 0xff00, "bra.b %s",
/* A: 0x60c0 */ BRANCH, 0, 0x60c0, 0xff00, "bra.b %s",
/* A: 0x6100 */ BRANCH, 0, 0x6100, 0xff00, "bsr.b %s",
/* A: 0x6140 */ BRANCH, 0, 0x6140, 0xff00, "bsr.b %s",
/* A: 0x6180 */ BRANCH, 0, 0x6180, 0xff00, "bsr.b %s",
/* A: 0x61c0 */ BRANCH, 0, 0x61c0, 0xff00, "bsr.b %s",
/* A: 0x6200 */ BRANCH, 0, 0x6200, 0xff00, "bhi.b %s",
/* A: 0x6240 */ BRANCH, 0, 0x6240, 0xff00, "bhi.b %s",
/* A: 0x6280 */ BRANCH, 0, 0x6280, 0xff00, "bhi.b %s",
/* A: 0x62c0 */ BRANCH, 0, 0x62c0, 0xff00, "bhi.b %s",
/* A: 0x6300 */ BRANCH, 0, 0x6300, 0xff00, "bls.b %s",
/* A: 0x6340 */ BRANCH, 0, 0x6340, 0xff00, "bls.b %s",
/* A: 0x6380 */ BRANCH, 0, 0x6380, 0xff00, "bls.b %s",
/* A: 0x63c0 */ BRANCH, 0, 0x63c0, 0xff00, "bls.b %s",
/* A: 0x6400 */ BRANCH, 0, 0x6400, 0xff00, "bcc.b %s",
/* A: 0x6440 */ BRANCH, 0, 0x6440, 0xff00, "bcc.b %s",
/* A: 0x6480 */ BRANCH, 0, 0x6480, 0xff00, "bcc.b %s",
/* A: 0x64c0 */ BRANCH, 0, 0x64c0, 0xff00, "bcc.b %s",
/* A: 0x6500 */ BRANCH, 0, 0x6500, 0xff00, "bcs.b %s",
/* A: 0x6540 */ BRANCH, 0, 0x6540, 0xff00, "bcs.b %s",
/* A: 0x6580 */ BRANCH, 0, 0x6580, 0xff00, "bcs.b %s",
/* A: 0x65c0 */ BRANCH, 0, 0x65c0, 0xff00, "bcs.b %s",
/* A: 0x6600 */ BRANCH, 0, 0x6600, 0xff00, "bne.b %s",
/* A: 0x6640 */ BRANCH, 0, 0x6640, 0xff00, "bne.b %s",
/* A: 0x6680 */ BRANCH, 0, 0x6680, 0xff00, "bne.b %s",
/* A: 0x66c0 */ BRANCH, 0, 0x66c0, 0xff00, "bne.b %s",
/* A: 0x6700 */ BRANCH, 0, 0x6700, 0xff00, "beq.b %s",
/* A: 0x6740 */ BRANCH, 0, 0x6740, 0xff00, "beq.b %s",

```


CHAPTER 10

```

/* A: 0x6780 */ BRANCH, 0, 0x6780, 0xff00, "beq.b %s",
/* A: 0x67c0 */ BRANCH, 0, 0x67c0, 0xff00, "beq.b %s",
/* A: 0x6800 */ BRANCH, 0, 0x6800, 0xff00, "bvc.b %s",
/* A: 0x6840 */ BRANCH, 0, 0x6840, 0xff00, "bvc.b %s",
/* A: 0x6880 */ BRANCH, 0, 0x6880, 0xff00, "bvc.b %s",
/* A: 0x68c0 */ BRANCH, 0, 0x68c0, 0xff00, "bvc.b %s",
/* A: 0x6900 */ BRANCH, 0, 0x6900, 0xff00, "bvs.b %s",
/* A: 0x6940 */ BRANCH, 0, 0x6940, 0xff00, "bvs.b %s",
/* A: 0x6980 */ BRANCH, 0, 0x6980, 0xff00, "bvs.b %s",
/* A: 0x69c0 */ BRANCH, 0, 0x69c0, 0xff00, "bvs.b %s",
/* A: 0x6a00 */ BRANCH, 0, 0x6a00, 0xff00, "bpl.b %s",
/* A: 0x6a40 */ BRANCH, 0, 0x6a40, 0xff00, "bpl.b %s",
/* A: 0x6a80 */ BRANCH, 0, 0x6a80, 0xff00, "bpl.b %s",
/* A: 0x6ac0 */ BRANCH, 0, 0x6ac0, 0xff00, "bpl.b %s",
/* A: 0x6b00 */ BRANCH, 0, 0x6b00, 0xff00, "bmi.b %s",
/* A: 0x6b40 */ BRANCH, 0, 0x6b40, 0xff00, "bmi.b %s",
/* A: 0x6b80 */ BRANCH, 0, 0x6b80, 0xff00, "bmi.b %s",
/* A: 0x6bc0 */ BRANCH, 0, 0x6bc0, 0xff00, "bmi.b %s",
/* A: 0x6c00 */ BRANCH, 0, 0x6c00, 0xff00, "bge.b %s",
/* A: 0x6c40 */ BRANCH, 0, 0x6c40, 0xff00, "bge.b %s",
/* A: 0x6c80 */ BRANCH, 0, 0x6c80, 0xff00, "bge.b %s",
/* A: 0x6cc0 */ BRANCH, 0, 0x6cc0, 0xff00, "bge.b %s",
/* A: 0x6d00 */ BRANCH, 0, 0x6d00, 0xff00, "bit.b %s",
/* A: 0x6d40 */ BRANCH, 0, 0x6d40, 0xff00, "bit.b %s",
/* A: 0x6d80 */ BRANCH, 0, 0x6d80, 0xff00, "bit.b %s",
/* A: 0x6dc0 */ BRANCH, 0, 0x6dc0, 0xff00, "bit.b %s",
/* A: 0x6e00 */ BRANCH, 0, 0x6e00, 0xff00, "bgt.b %s",
/* A: 0x6e40 */ BRANCH, 0, 0x6e40, 0xff00, "bgt.b %s",
/* A: 0x6e80 */ BRANCH, 0, 0x6e80, 0xff00, "bgt.b %s",
/* A: 0x6ec0 */ BRANCH, 0, 0x6ec0, 0xff00, "bgt.b %s",
/* A: 0x6f00 */ BRANCH, 0, 0x6f00, 0xff00, "ble.b %s",
/* A: 0x6f40 */ BRANCH, 0, 0x6f40, 0xff00, "ble.b %s",
/* A: 0x6f80 */ BRANCH, 0, 0x6f80, 0xff00, "ble.b %s",
/* A: 0x6fc0 */ BRANCH, 0, 0x6fc0, 0xff00, "ble.b %s",
/* A: 0x7000 */ MOVE_Q, 0, 0x7000, 0xff00, "movq.1 %s,D0",
/* A: 0x7040 */ MOVE_Q, 0, 0x7040, 0xff00, "movq.1 %s,D0",
/* A: 0x7080 */ MOVE_Q, 0, 0x7080, 0xff00, "movq.1 %s,D0",
/* A: 0x70c0 */ MOVE_Q, 0, 0x70c0, 0xff00, "movq.1 %s,D0",
/* A: 0x7100 */ MOVE_Q, 0, 0x7100, 0xff00, "movq.1 %s,D0",
/* A: 0x7140 */ MOVE_Q, 0, 0x7140, 0xff00, "movq.1 %s,D0",
/* A: 0x7180 */ MOVE_Q, 0, 0x7180, 0xff00, "movq.1 %s,D0",
/* A: 0x71c0 */ MOVE_Q, 0, 0x71c0, 0xff00, "movq.1 %s,D0",
/* A: 0x7200 */ MOVE_Q, 0, 0x7200, 0xff00, "movq.1 %s,D1",
/* A: 0x7240 */ MOVE_Q, 0, 0x7240, 0xff00, "movq.1 %s,D1",
/* A: 0x7280 */ MOVE_Q, 0, 0x7280, 0xff00, "movq.1 %s,D1",
/* A: 0x72c0 */ MOVE_Q, 0, 0x72c0, 0xff00, "movq.1 %s,D1",
/* A: 0x7300 */ MOVE_Q, 0, 0x7300, 0xff00, "movq.1 %s,D1",
/* A: 0x7340 */ MOVE_Q, 0, 0x7340, 0xff00, "movq.1 %s,D1",
/* A: 0x7380 */ MOVE_Q, 0, 0x7380, 0xff00, "movq.1 %s,D1",
/* A: 0x73c0 */ MOVE_Q, 0, 0x73c0, 0xff00, "movq.1 %s,D1",
/* A: 0x7400 */ MOVE_Q, 0, 0x7400, 0xff00, "movq.1 %s,D2",
/* A: 0x7440 */ MOVE_Q, 0, 0x7440, 0xff00, "movq.1 %s,D2",
/* A: 0x7480 */ MOVE_Q, 0, 0x7480, 0xff00, "movq.1 %s,D2",
/* A: 0x74c0 */ MOVE_Q, 0, 0x74c0, 0xff00, "movq.1 %s,D2",
/* A: 0x7500 */ MOVE_Q, 0, 0x7500, 0xff00, "movq.1 %s,D2",
/* A: 0x7540 */ MOVE_Q, 0, 0x7540, 0xff00, "movq.1 %s,D2",
/* A: 0x7580 */ MOVE_Q, 0, 0x7580, 0xff00, "movq.1 %s,D2",
/* A: 0x75c0 */ MOVE_Q, 0, 0x75c0, 0xff00, "movq.1 %s,D2",
/* A: 0x7600 */ MOVE_Q, 0, 0x7600, 0xff00, "movq.1 %s,D3",
/* A: 0x7640 */ MOVE_Q, 0, 0x7640, 0xff00, "movq.1 %s,D3",
/* A: 0x7680 */ MOVE_Q, 0, 0x7680, 0xff00, "movq.1 %s,D3",
/* A: 0x76c0 */ MOVE_Q, 0, 0x76c0, 0xff00, "movq.1 %s,D3",
/* A: 0x7700 */ MOVE_Q, 0, 0x7700, 0xff00, "movq.1 %s,D3",
/* A: 0x7740 */ MOVE_Q, 0, 0x7740, 0xff00, "movq.1 %s,D3",
/* A: 0x7780 */ MOVE_Q, 0, 0x7780, 0xff00, "movq.1 %s,D3",
/* A: 0x77c0 */ MOVE_Q, 0, 0x77c0, 0xff00, "movq.1 %s,D3",

```



```

/* A: 0x7800 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7840 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7880 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x78c0 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7900 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7940 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7980 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x79c0 */ MOVE_Q, 0, 0x7800, 0xff00, "movq.1 %s,D4",
/* A: 0x7a00 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7a40 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7a80 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7ac0 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7b00 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7b40 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7b80 */ MOVE_Q, 0, 0x7a00, 0xff00, "movq.1 %s,D5",
/* A: 0x7bc0 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7c00 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7c40 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7c80 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7cc0 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7d00 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7d40 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7d80 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7dc0 */ MOVE_Q, 0, 0x7c00, 0xff00, "movq.1 %s,D6",
/* A: 0x7e00 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7e40 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7e80 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7ec0 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7f00 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7f40 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7f80 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x7fc0 */ MOVE_Q, 0, 0x7e00, 0xff00, "movq.1 %s,D7",
/* A: 0x8000 */ EFFADD, 0, 0x8000, 0xffc0, "or.b %s,D0",
/* A: 0x8040 */ EFFADD, 0, 0x8040, 0xffc0, "or.w %s,D0",
/* A: 0x8080 */ EFFADD, 0, 0x8080, 0xffc0, "or.l %s,D0",
/* A: 0x80c0 */ EFFADD, 0, 0x80c0, 0xffc0, "divs.1 %s,D0",
/* A: 0x8100 */ EFFADD, 1, 0x8100, 0xffc0, "or.b D0,%s",
/* A: 0x8140 */ EFFADD, 0, 0x8140, 0xffc0, "or.w D0,%s",
/* A: 0x8180 */ EFFADD, 0, 0x8180, 0xffc0, "or.l D0,%s",
/* A: 0x81c0 */ EFFADD, 0, 0x81c0, 0xffc0, "divu.1 %s,D0",
/* A: 0x8200 */ EFFADD, 0, 0x8200, 0xffc0, "or.b %s,D1",
/* A: 0x8240 */ EFFADD, 0, 0x8240, 0xffc0, "or.w %s,D1",
/* A: 0x8280 */ EFFADD, 0, 0x8280, 0xffc0, "or.l %s,D1",
/* A: 0x82c0 */ EFFADD, 0, 0x82c0, 0xffc0, "divs.1 %s,D1",
/* A: 0x8300 */ EFFADD, 1, 0x8300, 0xffc0, "or.b D1,%s",
/* A: 0x8340 */ EFFADD, 0, 0x8340, 0xffc0, "or.w D1,%s",
/* A: 0x8380 */ EFFADD, 0, 0x8380, 0xffc0, "or.l D1,%s",
/* A: 0x83c0 */ EFFADD, 0, 0x83c0, 0xffc0, "divu.1 %s,D1",
/* A: 0x8400 */ EFFADD, 0, 0x8400, 0xffc0, "or.b %s,D2",
/* A: 0x8440 */ EFFADD, 0, 0x8440, 0xffc0, "or.w %s,D2",
/* A: 0x8480 */ EFFADD, 0, 0x8480, 0xffc0, "or.l %s,D2",
/* A: 0x84c0 */ EFFADD, 0, 0x84c0, 0xffc0, "divs.1 %s,D2",
/* A: 0x8500 */ EFFADD, 1, 0x8500, 0xffc0, "or.b D2,%s",
/* A: 0x8540 */ EFFADD, 0, 0x8540, 0xffc0, "or.w D2,%s",
/* A: 0x8580 */ EFFADD, 0, 0x8580, 0xffc0, "or.l D2,%s",
/* A: 0x85c0 */ EFFADD, 0, 0x85c0, 0xffc0, "divu.1 %s,D2",
/* A: 0x8600 */ EFFADD, 0, 0x8600, 0xffc0, "or.b %s,D3",
/* A: 0x8640 */ EFFADD, 0, 0x8640, 0xffc0, "or.w %s,D3",
/* A: 0x8680 */ EFFADD, 0, 0x8680, 0xffc0, "or.l %s,D3",
/* A: 0x86c0 */ EFFADD, 0, 0x86c0, 0xffc0, "divs.1 %s,D3",
/* A: 0x8700 */ EFFADD, 1, 0x8700, 0xffc0, "or.b D3,%s",
/* A: 0x8740 */ EFFADD, 0, 0x8740, 0xffc0, "or.w D3,%s",
/* A: 0x8780 */ EFFADD, 0, 0x8780, 0xffc0, "or.l D3,%s",
/* A: 0x87c0 */ EFFADD, 0, 0x87c0, 0xffc0, "divu.1 %s,D3",
/* A: 0x8800 */ EFFADD, 0, 0x8800, 0xffc0, "or.b %s,D4",
/* A: 0x8840 */ EFFADD, 0, 0x8840, 0xffc0, "or.w %s,D4",
/* A: 0x8880 */ EFFADD, 0, 0x8880, 0xffc0, "or.l %s,D4",

```

CHAPTER 10

```

/* A: 0x88c0 */ EFFADD, 0, 0x88c0, 0xffc0, "divs.1 %s,D4",
/* A: 0x8900 */ EFFADD, 1, 0x8900, 0xffc0, "or.b D4,%s",
/* A: 0x8940 */ EFFADD, 0, 0x8940, 0xffc0, "or.w D4,%s",
/* A: 0x8980 */ EFFADD, 0, 0x8980, 0xffc0, "or.l D4,%s",
/* A: 0x89c0 */ EFFADD, 0, 0x89c0, 0xffc0, "divu.1 %s,D4",
/* A: 0x8a00 */ EFFADD, 0, 0x8a00, 0xffc0, "or.b %s,D5",
/* A: 0x8a40 */ EFFADD, 0, 0x8a40, 0xffc0, "or.w %s,D5",
/* A: 0x8a80 */ EFFADD, 0, 0x8a80, 0xffc0, "or.l %s,D5",
/* A: 0x8ac0 */ EFFADD, 0, 0x8ac0, 0xffc0, "divs.1 %s,D5",
/* A: 0x8b00 */ EFFADD, 1, 0x8b00, 0xffc0, "or.b D5,%s",
/* A: 0x8b40 */ EFFADD, 0, 0x8b40, 0xffc0, "or.w D5,%s",
/* A: 0x8b80 */ EFFADD, 0, 0x8b80, 0xffc0, "or.l D5,%s",
/* A: 0x8bc0 */ EFFADD, 0, 0x8bc0, 0xffc0, "divu.1 %s,D5",
/* A: 0x8c00 */ EFFADD, 0, 0x8c00, 0xffc0, "or.b %s,D6",
/* A: 0x8c40 */ EFFADD, 0, 0x8c40, 0xffc0, "or.w %s,D6",
/* A: 0x8c80 */ EFFADD, 0, 0x8c80, 0xffc0, "or.l %s,D6",
/* A: 0x8cc0 */ EFFADD, 0, 0x8cc0, 0xffc0, "divs.1 %s,D6",
/* A: 0x8d00 */ EFFADD, 1, 0x8d00, 0xffc0, "or.b D6,%s",
/* A: 0x8d40 */ EFFADD, 0, 0x8d40, 0xffc0, "or.w D6,%s",
/* A: 0x8d80 */ EFFADD, 0, 0x8d80, 0xffc0, "or.l D6,%s",
/* A: 0x8dc0 */ EFFADD, 0, 0x8dc0, 0xffc0, "divu.1 %s,D6",
/* A: 0x8e00 */ EFFADD, 0, 0x8e00, 0xffc0, "or.b %s,D7",
/* A: 0x8e40 */ EFFADD, 0, 0x8e40, 0xffc0, "or.w %s,D7",
/* A: 0x8e80 */ EFFADD, 0, 0x8e80, 0xffc0, "or.l %s,D7",
/* A: 0x8ec0 */ EFFADD, 0, 0x8ec0, 0xffc0, "divs.1 %s,D7",
/* A: 0x8f00 */ EFFADD, 1, 0x8f00, 0xffc0, "or.b D7,%s",
/* A: 0x8f40 */ EFFADD, 0, 0x8f40, 0xffc0, "or.w D7,%s",
/* A: 0x8f80 */ EFFADD, 0, 0x8f80, 0xffc0, "or.l D7,%s",
/* A: 0x8fc0 */ EFFADD, 0, 0x8fc0, 0xffc0, "divu.1 %s,D7",
/* A: 0x9000 */ EFFADD, 0, 0x9000, 0xffc0, "sub.b %s,D0",
/* A: 0x9040 */ EFFADD, 0, 0x9040, 0xffc0, "sub.w %s,D0",
/* A: 0x9080 */ EFFADD, 0, 0x9080, 0xffc0, "sub.l %s,D0",
/* A: 0x90c0 */ ADDREA, 0, 0x90c0, 0xffc0, "suba.w %s,A0",
/* A: 0x9100 */ EFFADD, 1, 0x9100, 0xffc0, "sub.b D0,%s",
/* A: 0x9140 */ EFFADD, 1, 0x9140, 0xffc0, "sub.w D0,%s",
/* A: 0x9180 */ EFFADD, 1, 0x9180, 0xffc0, "sub.l D0,%s",
/* A: 0x91c0 */ ADDREA, 0, 0x91c0, 0xffc0, "suba.1 %s,A0",
/* A: 0x9200 */ EFFADD, 0, 0x9200, 0xffc0, "sub.b %s,D1",
/* A: 0x9240 */ EFFADD, 0, 0x9240, 0xffc0, "sub.w %s,D1",
/* A: 0x9280 */ EFFADD, 0, 0x9280, 0xffc0, "sub.l %s,D1",
/* A: 0x92c0 */ ADDREA, 0, 0x92c0, 0xffc0, "suba.w %s,A1",
/* A: 0x9300 */ EFFADD, 1, 0x9300, 0xffc0, "sub.b D1,%s",
/* A: 0x9340 */ EFFADD, 1, 0x9340, 0xffc0, "sub.w D1,%s",
/* A: 0x9380 */ EFFADD, 1, 0x9380, 0xffc0, "sub.l D1,%s",
/* A: 0x93c0 */ ADDREA, 0, 0x93c0, 0xffc0, "suba.1 %s,A1",
/* A: 0x9400 */ EFFADD, 0, 0x9400, 0xffc0, "sub.b %s,D2",
/* A: 0x9440 */ EFFADD, 0, 0x9440, 0xffc0, "sub.w %s,D2",
/* A: 0x9480 */ EFFADD, 0, 0x9480, 0xffc0, "sub.l %s,D2",
/* A: 0x94c0 */ ADDREA, 0, 0x94c0, 0xffc0, "suba.w %s,A2",
/* A: 0x9500 */ EFFADD, 1, 0x9500, 0xffc0, "sub.b D2,%s",
/* A: 0x9540 */ EFFADD, 1, 0x9540, 0xffc0, "sub.w D2,%s",
/* A: 0x9580 */ EFFADD, 1, 0x9580, 0xffc0, "sub.l D2,%s",
/* A: 0x95c0 */ ADDREA, 0, 0x95c0, 0xffc0, "suba.1 %s,A2",
/* A: 0x9600 */ EFFADD, 0, 0x9600, 0xffc0, "sub.b %s,D3",
/* A: 0x9640 */ EFFADD, 0, 0x9640, 0xffc0, "sub.w %s,D3",
/* A: 0x9680 */ EFFADD, 0, 0x9680, 0xffc0, "sub.l %s,D3",
/* A: 0x96c0 */ ADDREA, 0, 0x96c0, 0xffc0, "suba.w %s,A3",
/* A: 0x9700 */ EFFADD, 1, 0x9700, 0xffc0, "sub.b D3,%s",
/* A: 0x9740 */ EFFADD, 1, 0x9740, 0xffc0, "sub.w D3,%s",
/* A: 0x9780 */ EFFADD, 1, 0x9780, 0xffc0, "sub.l D3,%s",
/* A: 0x97c0 */ ADDREA, 0, 0x97c0, 0xffc0, "suba.1 %s,A3",
/* A: 0x9800 */ EFFADD, 0, 0x9800, 0xffc0, "sub.b %s,D4",
/* A: 0x9840 */ EFFADD, 0, 0x9840, 0xffc0, "sub.w %s,D4",
/* A: 0x9880 */ EFFADD, 0, 0x9880, 0xffc0, "sub.l %s,D4",
/* A: 0x98c0 */ ADDREA, 0, 0x98c0, 0xffc0, "suba.w %s,A4",
/* A: 0x9900 */ EFFADD, 1, 0x9900, 0xffc0, "sub.b D4,%s",

```

```

/* A: 0x9940 */ EFFADD, 1, 0x9940, 0xffc0, "sub.w D4,%s",
/* A: 0x9980 */ EFFADD, 1, 0x9980, 0xffc0, "sub.l D4,%s",
/* A: 0x99c0 */ ADDREA, 0, 0x99c0, 0xffc0, "suba.l %s,A4",
/* A: 0x9a00 */ EFFADD, 0, 0x9a00, 0xffc0, "sub.b %s,D5",
/* A: 0x9a40 */ EFFADD, 0, 0x9a40, 0xffc0, "sub.w %s,D5",
/* A: 0x9a80 */ EFFADD, 0, 0x9a80, 0xffc0, "sub.l %s,D5",
/* A: 0x9ac0 */ ADDREA, 0, 0x9ac0, 0xffc0, "suba.w %s,A5",
/* A: 0x9b00 */ EFFADD, 1, 0x9b00, 0xffc0, "sub.b D5,%s",
/* A: 0x9b40 */ EFFADD, 1, 0x9b40, 0xffc0, "sub.w D5,%s",
/* A: 0x9b80 */ EFFADD, 1, 0x9b80, 0xffc0, "sub.l D5,%s",
/* A: 0x9bc0 */ ADDREA, 0, 0x9bc0, 0xffc0, "suba.l %s,A5",
/* A: 0x9c00 */ EFFADD, 0, 0x9c00, 0xffc0, "sub.b %s,D6",
/* A: 0x9c40 */ EFFADD, 0, 0x9c40, 0xffc0, "sub.w %s,D6",
/* A: 0x9c80 */ EFFADD, 0, 0x9c80, 0xffc0, "sub.l %s,D6",
/* A: 0x9cc0 */ ADDREA, 0, 0x9cc0, 0xffc0, "suba.w %s,FP",
/* A: 0x9d00 */ EFFADD, 1, 0x9d00, 0xffc0, "sub.b D6,%s",
/* A: 0x9d40 */ EFFADD, 1, 0x9d40, 0xffc0, "sub.w D6,%s",
/* A: 0x9d80 */ EFFADD, 1, 0x9d80, 0xffc0, "sub.l D6,%s",
/* A: 0x9dc0 */ ADDREA, 0, 0x9dc0, 0xffc0, "suba.l %s,FP",
/* A: 0x9e00 */ EFFADD, 0, 0x9e00, 0xffc0, "sub.b %s,D7",
/* A: 0x9e40 */ EFFADD, 0, 0x9e40, 0xffc0, "sub.w %s,D7",
/* A: 0x9e80 */ EFFADD, 0, 0x9e80, 0xffc0, "sub.l %s,D7",
/* A: 0x9ec0 */ ADDREA, 0, 0x9ec0, 0xffc0, "suba.w %s,SP",
/* A: 0x9f00 */ EFFADD, 1, 0x9f00, 0xffc0, "sub.b D7,%s",
/* A: 0x9f40 */ EFFADD, 1, 0x9f40, 0xffc0, "sub.w D7,%s",
/* A: 0x9f80 */ EFFADD, 1, 0x9f80, 0xffc0, "sub.l D7,%s",
/* A: 0x9fc0 */ ADDREA, 0, 0x9fc0, 0xffc0, "suba.l %s,SP",
/* A: 0xa000 */ LINE_A, 0, 0xa000, 0xf000, "line A %s",
/* A: 0xa040 */ LINE_A, 0, 0xa040, 0xf000, "line A %s",
/* A: 0xa080 */ LINE_A, 0, 0xa080, 0xf000, "line A %s",
/* A: 0xa0c0 */ LINE_A, 0, 0xa0c0, 0xf000, "line A %s",
/* A: 0xa100 */ LINE_A, 0, 0xa100, 0xf000, "line A %s",
/* A: 0xa140 */ LINE_A, 0, 0xa140, 0xf000, "line A %s",
/* A: 0xa180 */ LINE_A, 0, 0xa180, 0xf000, "line A %s",
/* A: 0xa1c0 */ LINE_A, 0, 0xa1c0, 0xf000, "line A %s",
/* A: 0xa200 */ LINE_A, 0, 0xa200, 0xf000, "line A %s",
/* A: 0xa240 */ LINE_A, 0, 0xa240, 0xf000, "line A %s",
/* A: 0xa280 */ LINE_A, 0, 0xa280, 0xf000, "line A %s",
/* A: 0xa2c0 */ LINE_A, 0, 0xa2c0, 0xf000, "line A %s",
/* A: 0xa300 */ LINE_A, 0, 0xa300, 0xf000, "line A %s",
/* A: 0xa340 */ LINE_A, 0, 0xa340, 0xf000, "line A %s",
/* A: 0xa380 */ LINE_A, 0, 0xa380, 0xf000, "line A %s",
/* A: 0xa3c0 */ LINE_A, 0, 0xa3c0, 0xf000, "line A %s",
/* A: 0xa400 */ LINE_A, 0, 0xa400, 0xf000, "line A %s",
/* A: 0xa440 */ LINE_A, 0, 0xa440, 0xf000, "line A %s",
/* A: 0xa480 */ LINE_A, 0, 0xa480, 0xf000, "line A %s",
/* A: 0xa4c0 */ LINE_A, 0, 0xa4c0, 0xf000, "line A %s",
/* A: 0xa500 */ LINE_A, 0, 0xa500, 0xf000, "line A %s",
/* A: 0xa540 */ LINE_A, 0, 0xa540, 0xf000, "line A %s",
/* A: 0xa580 */ LINE_A, 0, 0xa580, 0xf000, "line A %s",
/* A: 0xa5c0 */ LINE_A, 0, 0xa5c0, 0xf000, "line A %s",
/* A: 0xa600 */ LINE_A, 0, 0xa600, 0xf000, "line A %s",
/* A: 0xa640 */ LINE_A, 0, 0xa640, 0xf000, "line A %s",
/* A: 0xa680 */ LINE_A, 0, 0xa680, 0xf000, "line A %s",
/* A: 0xa6c0 */ LINE_A, 0, 0xa6c0, 0xf000, "line A %s",
/* A: 0xa700 */ LINE_A, 0, 0xa700, 0xf000, "line A %s",
/* A: 0xa740 */ LINE_A, 0, 0xa740, 0xf000, "line A %s",
/* A: 0xa780 */ LINE_A, 0, 0xa780, 0xf000, "line A %s",
/* A: 0xa7c0 */ LINE_A, 0, 0xa7c0, 0xf000, "line A %s",
/* A: 0xa800 */ LINE_A, 0, 0xa800, 0xf000, "line A %s",
/* A: 0xa840 */ LINE_A, 0, 0xa840, 0xf000, "line A %s",
/* A: 0xa880 */ LINE_A, 0, 0xa880, 0xf000, "line A %s",
/* A: 0xa8c0 */ LINE_A, 0, 0xa8c0, 0xf000, "line A %s",
/* A: 0xa900 */ LINE_A, 0, 0xa900, 0xf000, "line A %s",
/* A: 0xa940 */ LINE_A, 0, 0xa940, 0xf000, "line A %s",
/* A: 0xa980 */ LINE_A, 0, 0xa980, 0xf000, "line A %s",
/* A: 0xa9c0 */ LINE_A, 0, 0xa9c0, 0xf000, "line A %s",

```


CHAPTER 10

```

/* A: 0xaa00 */ LINE_A, 0, 0xaa00, 0xf000, "line A %s",
/* A: 0xaa40 */ LINE_A, 0, 0xaa40, 0xf000, "line A %s",
/* A: 0xaa80 */ LINE_A, 0, 0xaa80, 0xf000, "line A %s",
/* A: 0xaac0 */ LINE_A, 0, 0xaac0, 0xf000, "line A %s",
/* A: 0xab00 */ LINE_A, 0, 0xab00, 0xf000, "line A %s",
/* A: 0xab40 */ LINE_A, 0, 0xab40, 0xf000, "line A %s",
/* A: 0xab80 */ LINE_A, 0, 0xab80, 0xf000, "line A %s",
/* A: 0xabc0 */ LINE_A, 0, 0xabc0, 0xf000, "line A %s",
/* A: 0xac00 */ LINE_A, 0, 0xac00, 0xf000, "line A %s",
/* A: 0xac40 */ LINE_A, 0, 0xac40, 0xf000, "line A %s",
/* A: 0xac80 */ LINE_A, 0, 0xac80, 0xf000, "line A %s",
/* A: 0xacc0 */ LINE_A, 0, 0xacc0, 0xf000, "line A %s",
/* A: 0xad00 */ LINE_A, 0, 0xad00, 0xf000, "line A %s",
/* A: 0xad40 */ LINE_A, 0, 0xad40, 0xf000, "line A %s",
/* A: 0xad80 */ LINE_A, 0, 0xad80, 0xf000, "line A %s",
/* A: 0xadc0 */ LINE_A, 0, 0xadc0, 0xf000, "line A %s",
/* A: 0xae00 */ LINE_A, 0, 0xae00, 0xf000, "line A %s",
/* A: 0xae40 */ LINE_A, 0, 0xae40, 0xf000, "line A %s",
/* A: 0xae80 */ LINE_A, 0, 0xae80, 0xf000, "line A %s",
/* A: 0xaec0 */ LINE_A, 0, 0xaec0, 0xf000, "line A %s",
/* A: 0xaf00 */ LINE_A, 0, 0xaf00, 0xf000, "line A %s",
/* A: 0xaf40 */ LINE_A, 0, 0xaf40, 0xf000, "line A %s",
/* A: 0xaf80 */ LINE_A, 0, 0xaf80, 0xf000, "line A %s",
/* A: 0xafc0 */ LINE_A, 0, 0xafc0, 0xf000, "line A %s",
/* A: 0xb000 */ EFFADD, 0, 0xb000, 0xffc0, "eor.b D0,%s",
/* A: 0xb040 */ EFFADD, 0, 0xb040, 0xffc0, "eor.w D0,%s",
/* A: 0xb080 */ EFFADD, 0, 0xb080, 0xffc0, "eor.l D0,%s",
/* A: 0xb0c0 */ ADDREA, 0, 0xb0c0, 0xffc0, "cmpa.w %s,A0",
/* A: 0xb100 */ EFFADD, 1, 0xb100, 0xffc0, "cmp.b D0,%s",
/* A: 0xb140 */ EFFADD, 1, 0xb140, 0xffc0, "cmp.w D0,%s",
/* A: 0xb180 */ EFFADD, 1, 0xb180, 0xffc0, "cmp.l D0,%s",
/* A: 0xb1c0 */ ADDREA, 0, 0xb1c0, 0xffc0, "cmpa.l %s,A0",
/* A: 0xb200 */ EFFADD, 0, 0xb200, 0xffc0, "eor.b D1,%s",
/* A: 0xb240 */ EFFADD, 0, 0xb240, 0xffc0, "eor.w D1,%s",
/* A: 0xb280 */ EFFADD, 0, 0xb280, 0xffc0, "eor.l D1,%s",
/* A: 0xb2c0 */ ADDREA, 0, 0xb2c0, 0xffc0, "cmpa.w %s,A1",
/* A: 0xb300 */ EFFADD, 1, 0xb300, 0xffc0, "cmp.b D1,%s",
/* A: 0xb340 */ EFFADD, 1, 0xb340, 0xffc0, "cmp.w D1,%s",
/* A: 0xb380 */ EFFADD, 1, 0xb380, 0xffc0, "cmp.l D1,%s",
/* A: 0xb3c0 */ ADDREA, 0, 0xb3c0, 0xffc0, "cmpa.l %s,A1",
/* A: 0xb400 */ EFFADD, 0, 0xb400, 0xffc0, "eor.b D2,%s",
/* A: 0xb440 */ EFFADD, 0, 0xb440, 0xffc0, "eor.w D2,%s",
/* A: 0xb480 */ EFFADD, 0, 0xb480, 0xffc0, "eor.l D2,%s",
/* A: 0xb4c0 */ ADDREA, 0, 0xb4c0, 0xffc0, "cmpa.w %s,A2",
/* A: 0xb500 */ EFFADD, 1, 0xb500, 0xffc0, "cmp.b D2,%s",
/* A: 0xb540 */ EFFADD, 1, 0xb540, 0xffc0, "cmp.w D2,%s",
/* A: 0xb580 */ EFFADD, 1, 0xb580, 0xffc0, "cmp.l D2,%s",
/* A: 0xb5c0 */ ADDREA, 0, 0xb5c0, 0xffc0, "cmpa.l %s,A2",
/* A: 0xb600 */ EFFADD, 0, 0xb600, 0xffc0, "eor.b D3,%s",
/* A: 0xb640 */ EFFADD, 0, 0xb640, 0xffc0, "eor.w D3,%s",
/* A: 0xb680 */ EFFADD, 0, 0xb680, 0xffc0, "eor.l D3,%s",
/* A: 0xb6c0 */ ADDREA, 0, 0xb6c0, 0xffc0, "cmpa.w %s,A3",
/* A: 0xb700 */ EFFADD, 1, 0xb700, 0xffc0, "cmp.b D3,%s",
/* A: 0xb740 */ EFFADD, 1, 0xb740, 0xffc0, "cmp.w D3,%s",
/* A: 0xb780 */ EFFADD, 1, 0xb780, 0xffc0, "cmp.l D3,%s",
/* A: 0xb7c0 */ ADDREA, 0, 0xb7c0, 0xffc0, "cmpa.l %s,A3",
/* A: 0xb800 */ EFFADD, 0, 0xb800, 0xffc0, "eor.b D4,%s",
/* A: 0xb840 */ EFFADD, 0, 0xb840, 0xffc0, "eor.w D4,%s",
/* A: 0xb880 */ EFFADD, 0, 0xb880, 0xffc0, "eor.l D4,%s",
/* A: 0xb8c0 */ ADDREA, 0, 0xb8c0, 0xffc0, "cmpa.w %s,A4",
/* A: 0xb900 */ EFFADD, 1, 0xb900, 0xffc0, "cmp.b D4,%s",
/* A: 0xb940 */ EFFADD, 1, 0xb940, 0xffc0, "cmp.w D4,%s",
/* A: 0xb980 */ EFFADD, 1, 0xb980, 0xffc0, "cmp.l D4,%s",
/* A: 0xb9c0 */ ADDREA, 0, 0xb9c0, 0xffc0, "cmpa.l %s,A4",
/* A: 0xba00 */ EFFADD, 0, 0xba00, 0xffc0, "eor.b D5,%s",
/* A: 0xba40 */ EFFADD, 0, 0xba40, 0xffc0, "eor.w D5,%s",
/* A: 0xba80 */ EFFADD, 0, 0xba80, 0xffc0, "eor.l D5,%s",

```



```

/* A: 0xbac0 */ ADDREA, 0,
/* A: 0xbb00 */ EFFADD, 1,
/* A: 0xbb40 */ EFFADD, 1,
/* A: 0xbb80 */ EFFADD, 1,
/* A: 0xbbc0 */ ADDREA, 0,
/* A: 0xbc00 */ EFFADD, 0,
/* A: 0xbc40 */ EFFADD, 0,
/* A: 0xbc80 */ EFFADD, 0,
/* A: 0bcc00 */ ADDREA, 0,
/* A: 0xbd00 */ EFFADD, 1,
/* A: 0xbd40 */ EFFADD, 1,
/* A: 0xbd80 */ EFFADD, 1,
/* A: 0xbdc0 */ ADDREA, 0,
/* A: 0xbe00 */ EFFADD, 0,
/* A: 0xbe40 */ EFFADD, 0,
/* A: 0xbe80 */ EFFADD, 0,
/* A: 0bec00 */ ADDREA, 0,
/* A: 0xbf00 */ EFFADD, 1,
/* A: 0xbf40 */ EFFADD, 1,
/* A: 0xbf80 */ EFFADD, 1,
/* A: 0bfc00 */ ADDREA, 0,
/* A: 0xc000 */ EFFADD, 0,
/* A: 0xc040 */ EFFADD, 0,
/* A: 0xc080 */ EFFADD, 0,
/* A: 0xc0c0 */ EFFADD, 0,
/* A: 0xc100 */ EFFADD, 1,
/* A: 0xc140 */ EFFADD, 2,
/* A: 0xc180 */ EFFADD, 1,
/* A: 0xc1c0 */ EFFADD, 0,
/* A: 0xc200 */ EFFADD, 0,
/* A: 0xc240 */ EFFADD, 0,
/* A: 0xc280 */ EFFADD, 0,
/* A: 0xc2c0 */ EFFADD, 0,
/* A: 0xc300 */ EFFADD, 1,
/* A: 0xc340 */ EFFADD, 2,
/* A: 0xc380 */ EFFADD, 1,
/* A: 0xc3c0 */ EFFADD, 0,
/* A: 0xc400 */ EFFADD, 0,
/* A: 0xc440 */ EFFADD, 0,
/* A: 0xc480 */ EFFADD, 0,
/* A: 0xc4c0 */ EFFADD, 0,
/* A: 0xc500 */ EFFADD, 1,
/* A: 0xc540 */ EFFADD, 2,
/* A: 0xc580 */ EFFADD, 1,
/* A: 0xc5c0 */ EFFADD, 0,
/* A: 0xc600 */ EFFADD, 0,
/* A: 0xc640 */ EFFADD, 0,
/* A: 0xc680 */ EFFADD, 0,
/* A: 0xc6c0 */ EFFADD, 0,
/* A: 0xc700 */ EFFADD, 1,
/* A: 0xc740 */ EFFADD, 2,
/* A: 0xc780 */ EFFADD, 1,
/* A: 0xc7c0 */ EFFADD, 0,
/* A: 0xc800 */ EFFADD, 0,
/* A: 0xc840 */ EFFADD, 0,
/* A: 0xc880 */ EFFADD, 0,
/* A: 0xc8c0 */ EFFADD, 0,
/* A: 0xc900 */ EFFADD, 1,
/* A: 0xc940 */ EFFADD, 2,
/* A: 0xc980 */ EFFADD, 1,
/* A: 0xc9c0 */ EFFADD, 0,
/* A: 0xca00 */ EFFADD, 0,
/* A: 0xca40 */ EFFADD, 0,
/* A: 0xca80 */ EFFADD, 0,
/* A: 0xcac0 */ EFFADD, 0,
/* A: 0xcb00 */ EFFADD, 1,
0xbac0, 0xffc0, "cmpa.w %s,A5",
0xbb00, 0xffc0, "cmp.b D5,%s",
0xbb40, 0xffc0, "cmp.w D5,%s",
0xbb80, 0xffc0, "cmp.l D5,%s",
0bbbc0, 0xffc0, "cmpa.l %s,A5",
0xbc00, 0xffc0, "eor.b D6,%s",
0xbc40, 0xffc0, "eor.w D6,%s",
0bc80, 0xffc0, "eor.l D6,%s",
0bcc0, 0xffc0, "cmpa.w %s,FP",
0bd00, 0xffc0, "cmp.b D6,%s",
0bd40, 0xffc0, "cmp.w D6,%s",
0bd80, 0xffc0, "cmp.l D6,%s",
0bdc0, 0xffc0, "cmpa.l %s,FP",
0be00, 0xffc0, "eor.b D7,%s",
0be40, 0xffc0, "eor.w D7,%s",
0be80, 0xffc0, "eor.l D7,%s",
0bec0, 0xffc0, "cmpa.w %s,SP",
0bf00, 0xffc0, "cmp.b D7,%s",
0bf40, 0xffc0, "cmp.w D7,%s",
0bf80, 0xffc0, "cmp.l D7,%s",
0bfc0, 0xffc0, "cmpa.l %s,SP",
0xc000, 0xffc0, "and.b %s,D0",
0xc040, 0xffc0, "and.w %s,D0",
0xc080, 0xffc0, "and.l %s,D0",
0xc0c0, 0xffc0, "mul.s.l %s,D0",
0xc100, 0xffc0, "and.b D0,%s",
0xc140, 0xffc0, "and.w D0,%s",
0xc180, 0xffc0, "and.l D0,%s",
0xc1c0, 0xffc0, "mul.u.l %s,D0",
0xc200, 0xffc0, "and.b %s,D1",
0xc240, 0xffc0, "and.w %s,D1",
0xc280, 0xffc0, "and.l %s,D1",
0xc2c0, 0xffc0, "mul.s.l %s,D1",
0xc300, 0xffc0, "and.b D1,%s",
0xc340, 0xffc0, "and.w D1,%s",
0xc380, 0xffc0, "and.l D1,%s",
0xc3c0, 0xffc0, "mul.u.l %s,D1",
0xc400, 0xffc0, "and.b %s,D2",
0xc440, 0xffc0, "and.w %s,D2",
0xc480, 0xffc0, "and.l %s,D2",
0xc4c0, 0xffc0, "mul.s.l %s,D2",
0xc500, 0xffc0, "and.b D2,%s",
0xc540, 0xffc0, "and.w D2,%s",
0xc580, 0xffc0, "and.l D2,%s",
0xc5c0, 0xffc0, "mul.u.l %s,D2",
0xc600, 0xffc0, "and.b %s,D3",
0xc640, 0xffc0, "and.w %s,D3",
0xc680, 0xffc0, "and.l %s,D3",
0xc6c0, 0xffc0, "mul.s.l %s,D3",
0xc700, 0xffc0, "and.b D3,%s",
0xc740, 0xffc0, "and.w D3,%s",
0xc780, 0xffc0, "and.l D3,%s",
0xc7c0, 0xffc0, "mul.u.l %s,D3",
0xc800, 0xffc0, "and.b %s,D4",
0xc840, 0xffc0, "and.w %s,D4",
0xc880, 0xffc0, "and.l %s,D4",
0xc8c0, 0xffc0, "mul.s.l %s,D4",
0xc900, 0xffc0, "and.b D4,%s",
0xc940, 0xffc0, "and.w D4,%s",
0xc980, 0xffc0, "and.l D4,%s",
0xc9c0, 0xffc0, "mul.u.l %s,D4",
0xca00, 0xffc0, "and.b %s,D5",
0xca40, 0xffc0, "and.w %s,D5",
0xca80, 0xffc0, "and.l %s,D5",
0xcac0, 0xffc0, "mul.s.l %s,D5",
0xcb00, 0xffc0, "and.b D5,%s",

```

CHAPTER 10

```

/* A: 0xcb40 */ EFFADD, 2, 0xcb40, 0xffc0, "and.w D5,%s",
/* A: 0xcb80 */ EFFADD, 1, 0xcb80, 0xffc0, "and.l D5,%s",
/* A: 0xcbc0 */ EFFADD, 0, 0xcbc0, 0xffc0, "mulu.1 %s,D5",
/* A: 0xcc00 */ EFFADD, 0, 0xcc00, 0xffc0, "and.b %s,D6",
/* A: 0xcc40 */ EFFADD, 0, 0xcc40, 0xffc0, "and.w %s,D6",
/* A: 0xcc80 */ EFFADD, 0, 0xcc80, 0xffc0, "and.l %s,D6",
/* A: 0xccc0 */ EFFADD, 0, 0xccc0, 0xffc0, "mul.s.1 %s,D6",
/* A: 0xcd00 */ EFFADD, 1, 0xcd00, 0xffc0, "and.b D6,%s",
/* A: 0xcd40 */ EFFADD, 2, 0xcd40, 0xffc0, "and.w D6,%s",
/* A: 0xcd80 */ EFFADD, 1, 0xcd80, 0xffc0, "and.l D6,%s",
/* A: 0xcdc0 */ EFFADD, 0, 0xcdc0, 0xffc0, "mulu.1 %s,D6",
/* A: 0xce00 */ EFFADD, 0, 0xce00, 0xffc0, "and.b %s,D7",
/* A: 0xce40 */ EFFADD, 0, 0xce40, 0xffc0, "and.w %s,D7",
/* A: 0xce80 */ EFFADD, 0, 0xce80, 0xffc0, "and.l %s,D7",
/* A: 0xcec0 */ EFFADD, 0, 0xcec0, 0xffc0, "mul.s.1 %s,D7",
/* A: 0xcf00 */ EFFADD, 1, 0xcf00, 0xffc0, "and.b D7,%s",
/* A: 0xcf40 */ EFFADD, 2, 0xcf40, 0xffc0, "and.w D7,%s",
/* A: 0xcf80 */ EFFADD, 1, 0xcf80, 0xffc0, "and.l D7,%s",
/* A: 0xcfc0 */ EFFADD, 0, 0xcfc0, 0xffc0, "mulu.1 %s,D7",
/* A: 0xd000 */ EFFADD, 0, 0xd000, 0xffc0, "add.b %s,D0",
/* A: 0xd040 */ EFFADD, 0, 0xd040, 0xffc0, "add.w %s,D0",
/* A: 0xd080 */ EFFADD, 0, 0xd080, 0xffc0, "add.l %s,D0",
/* A: 0xd0c0 */ ADDREA, 0, 0xd0c0, 0xffc0, "adda.w %s,A0",
/* A: 0xd100 */ EFFADD, 1, 0xd100, 0xffc0, "add.b D0,%s",
/* A: 0xd140 */ EFFADD, 1, 0xd140, 0xffc0, "add.w D0,%s",
/* A: 0xd180 */ EFFADD, 1, 0xd180, 0xffc0, "add.l D0,%s",
/* A: 0xd1c0 */ ADDREA, 0, 0xd1c0, 0xffc0, "adda.l %s,A0",
/* A: 0xd200 */ EFFADD, 0, 0xd200, 0xffc0, "add.b %s,D1",
/* A: 0xd240 */ EFFADD, 0, 0xd240, 0xffc0, "add.w %s,D1",
/* A: 0xd280 */ EFFADD, 0, 0xd280, 0xffc0, "add.l %s,D1",
/* A: 0xd2c0 */ ADDREA, 0, 0xd2c0, 0xffc0, "adda.w %s,A1",
/* A: 0xd300 */ EFFADD, 1, 0xd300, 0xffc0, "add.b D1,%s",
/* A: 0xd340 */ EFFADD, 1, 0xd340, 0xffc0, "add.w D1,%s",
/* A: 0xd380 */ EFFADD, 1, 0xd380, 0xffc0, "add.l D1,%s",
/* A: 0xd3c0 */ ADDREA, 0, 0xd3c0, 0xffc0, "adda.l %s,A1",
/* A: 0xd400 */ EFFADD, 0, 0xd400, 0xffc0, "add.b %s,D2",
/* A: 0xd440 */ EFFADD, 0, 0xd440, 0xffc0, "add.w %s,D2",
/* A: 0xd480 */ EFFADD, 0, 0xd480, 0xffc0, "add.l %s,D2",
/* A: 0xd4c0 */ ADDREA, 0, 0xd4c0, 0xffc0, "adda.w %s,A2",
/* A: 0xd500 */ EFFADD, 1, 0xd500, 0xffc0, "add.b D2,%s",
/* A: 0xd540 */ EFFADD, 1, 0xd540, 0xffc0, "add.w D2,%s",
/* A: 0xd580 */ EFFADD, 1, 0xd580, 0xffc0, "add.l D2,%s",
/* A: 0xd5c0 */ ADDREA, 0, 0xd5c0, 0xffc0, "adda.l %s,A2",
/* A: 0xd600 */ EFFADD, 0, 0xd600, 0xffc0, "add.b %s,D3",
/* A: 0xd640 */ EFFADD, 0, 0xd640, 0xffc0, "add.w %s,D3",
/* A: 0xd680 */ EFFADD, 0, 0xd680, 0xffc0, "add.l %s,D3",
/* A: 0xd6c0 */ ADDREA, 0, 0xd6c0, 0xffc0, "adda.w %s,A3",
/* A: 0xd700 */ EFFADD, 1, 0xd700, 0xffc0, "add.b D3,%s",
/* A: 0xd740 */ EFFADD, 1, 0xd740, 0xffc0, "add.w D3,%s",
/* A: 0xd780 */ EFFADD, 1, 0xd780, 0xffc0, "add.l D3,%s",
/* A: 0xd7c0 */ ADDREA, 0, 0xd7c0, 0xffc0, "adda.l %s,A3",
/* A: 0xd800 */ EFFADD, 0, 0xd800, 0xffc0, "add.b %s,D4",
/* A: 0xd840 */ EFFADD, 0, 0xd840, 0xffc0, "add.w %s,D4",
/* A: 0xd880 */ EFFADD, 0, 0xd880, 0xffc0, "add.l %s,D4",
/* A: 0xd8c0 */ ADDREA, 0, 0xd8c0, 0xffc0, "adda.w %s,A4",
/* A: 0xd900 */ EFFADD, 1, 0xd900, 0xffc0, "add.b D4,%s",
/* A: 0xd940 */ EFFADD, 1, 0xd940, 0xffc0, "add.w D4,%s",
/* A: 0xd980 */ EFFADD, 1, 0xd980, 0xffc0, "add.l D4,%s",
/* A: 0xd9c0 */ ADDREA, 0, 0xd9c0, 0xffc0, "adda.l %s,A4",
/* A: 0xda00 */ EFFADD, 0, 0xda00, 0xffc0, "add.b %s,D5",
/* A: 0xda40 */ EFFADD, 0, 0xda40, 0xffc0, "add.w %s,D5",
/* A: 0xda80 */ EFFADD, 0, 0xda80, 0xffc0, "add.l %s,D5",
/* A: 0xdac0 */ ADDREA, 0, 0xdac0, 0xffc0, "adda.w %s,A5",
/* A: 0xdb00 */ EFFADD, 1, 0xdb00, 0xffc0, "add.b D5,%s",
/* A: 0xdb40 */ EFFADD, 1, 0xdb40, 0xffc0, "add.w D5,%s",
/* A: 0xdb80 */ EFFADD, 1, 0xdb80, 0xffc0, "add.l D5,%s",
/* A: 0dbc0 */ ADDREA, 0, 0dbc0, 0xffc0, "adda.l %s,A5",

```

```

/* A: 0xdc00 */ EFFADD, 0,
/* A: 0xdc40 */ EFFADD, 0,
/* A: 0xdc80 */ EFFADD, 0,
/* A: 0xdcc0 */ ADDREA, 0,
/* A: 0xdd00 */ EFFADD, 1,
/* A: 0xdd40 */ EFFADD, 1,
/* A: 0xdd80 */ EFFADD, 1,
/* A: 0ddc0 */ ADDREA, 0,
/* A: 0xde00 */ EFFADD, 0,
/* A: 0xde40 */ EFFADD, 0,
/* A: 0xde80 */ EFFADD, 0,
/* A: 0xdec0 */ ADDREA, 0,
/* A: 0xdf00 */ EFFADD, 1,
/* A: 0xdf40 */ EFFADD, 1,
/* A: 0xdf80 */ EFFADD, 1,
/* A: 0xdfc0 */ ADDREA, 0,
/* A: 0xe000 */ SFTROT, 0,
/* A: 0xe040 */ SFTROT, 0,
/* A: 0xe080 */ SFTROT, 0,
/* A: 0xe0c0 */ EFFADD, 0,
/* A: 0xe100 */ SFTROT, 0,
/* A: 0xe140 */ SFTROT, 0,
/* A: 0xe180 */ SFTROT, 0,
/* A: 0xe1c0 */ EFFADD, 0,
/* A: 0xe200 */ SFTROT, 0,
/* A: 0xe240 */ SFTROT, 0,
/* A: 0xe280 */ SFTROT, 0,
/* A: 0xe2c0 */ EFFADD, 0,
/* A: 0xe300 */ SFTROT, 0,
/* A: 0xe340 */ SFTROT, 0,
/* A: 0xe380 */ SFTROT, 0,
/* A: 0xe3c0 */ EFFADD, 0,
/* A: 0xe400 */ SFTROT, 0,
/* A: 0xe440 */ SFTROT, 0,
/* A: 0xe480 */ SFTROT, 0,
/* A: 0xe4c0 */ EFFADD, 0,
/* A: 0xe500 */ SFTROT, 0,
/* A: 0xe540 */ SFTROT, 0,
/* A: 0xe580 */ SFTROT, 0,
/* A: 0xe5c0 */ EFFADD, 0,
/* A: 0xe600 */ SFTROT, 0,
/* A: 0xe640 */ SFTROT, 0,
/* A: 0xe680 */ SFTROT, 0,
/* A: 0xe6c0 */ EFFADD, 0,
/* A: 0xe700 */ SFTROT, 0,
/* A: 0xe740 */ SFTROT, 0,
/* A: 0xe780 */ SFTROT, 0,
/* A: 0xe7c0 */ EFFADD, 0,
/* A: 0xe800 */ SFTROT, 0,
/* A: 0xe840 */ SFTROT, 0,
/* A: 0xe880 */ SFTROT, 0,
/* A: 0xe8c0 */ NONE, 0,
/* A: 0xe900 */ SFTROT, 0,
/* A: 0xe940 */ SFTROT, 0,
/* A: 0xe980 */ SFTROT, 0,
/* A: 0xe9c0 */ NONE, 0,
/* A: 0xea00 */ SFTROT, 0,
/* A: 0xea40 */ SFTROT, 0,
/* A: 0xea80 */ SFTROT, 0,
/* A: 0xeac0 */ NONE, 0,
/* A: 0xeb00 */ SFTROT, 0,
/* A: 0xeb40 */ SFTROT, 0,
/* A: 0xeb80 */ SFTROT, 0,
/* A: 0xebc0 */ NONE, 0,
/* A: 0xec00 */ SFTROT, 0,
/* A: 0xec40 */ SFTROT, 0,
0xdc00, 0xffc0, "add.b %s,D6",
0xdc40, 0xffc0, "add.w %s,D6",
0xdc80, 0xffc0, "add.l %s,D6",
0xdcc0, 0xffc0, "adda.w %s,FP",
0xdd00, 0xffc0, "add.b D6,%s",
0xdd40, 0xffc0, "add.w D6,%s",
0xdd80, 0xffc0, "add.l D6,%s",
0ddc0, 0xffc0, "adda.l %s,FP",
0xde00, 0xffc0, "add.b %s,D7",
0xde40, 0xffc0, "add.w %s,D7",
0xde80, 0xffc0, "add.l %s,D7",
0xdec0, 0xffc0, "adda.w %s,SP",
0xdf00, 0xffc0, "add.b D7,%s",
0xdf40, 0xffc0, "add.w D7,%s",
0xdf80, 0xffc0, "add.l D7,%s",
0xdfc0, 0xffc0, "adda.l %s,SP",
0xe000, 0xffc0, "%sr.b %s",
0xe040, 0xffc0, "%sr.w %s",
0xe080, 0xffc0, "%sr.l %s",
0xe0c0, 0xffc0, "asr.l %s",
0xe100, 0xffc0, "%sl.b %s",
0xe140, 0xffc0, "%sl.w %s",
0xe180, 0xffc0, "%sl.l %s",
0xe1c0, 0xffc0, "asl.l %s",
0xe200, 0xffc0, "%sr.b %s",
0xe240, 0xffc0, "%sr.w %s",
0xe280, 0xffc0, "%sr.l %s",
0xe2c0, 0xffc0, "lsr.l %s",
0xe300, 0xffc0, "%sl.b %s",
0xe340, 0xffc0, "%sl.w %s",
0xe380, 0xffc0, "%sl.l %s",
0xe3c0, 0xffc0, "lsl.l %s",
0xe400, 0xffc0, "%sr.b %s",
0xe440, 0xffc0, "%sr.w %s",
0xe480, 0xffc0, "%sr.l %s",
0xe4c0, 0xffc0, "rorr.l %s",
0xe500, 0xffc0, "%sl.b %s",
0xe540, 0xffc0, "%sl.w %s",
0xe580, 0xffc0, "%sl.l %s",
0xe5c0, 0xffc0, "rorl.l %s",
0xe600, 0xffc0, "%sr.b %s",
0xe640, 0xffc0, "%sr.w %s",
0xe680, 0xffc0, "%sr.l %s",
0xe6c0, 0xffc0, "ror.l %s",
0xe700, 0xffc0, "%sl.b %s",
0xe740, 0xffc0, "%sl.w %s",
0xe780, 0xffc0, "%sl.l %s",
0xe7c0, 0xffc0, "rol.l %s",
0xe800, 0xffc0, "%sr.b %s",
0xe840, 0xffc0, "%sr.w %s",
0xe880, 0xffc0, "%sr.l %s",
0xe900, 0xffc0, "unimplemented",
0xe940, 0xffc0, "%sl.b %s",
0xe980, 0xffc0, "%sl.w %s",
0xe9c0, 0xffc0, "%sl.l %s",
0xea00, 0xffc0, "unimplemented",
0xea40, 0xffc0, "%sr.b %s",
0xea80, 0xffc0, "%sr.w %s",
0xeac0, 0xffc0, "%sr.l %s",
0xeb00, 0xffc0, "unimplemented",
0xeb40, 0xffc0, "%sl.b %s",
0xeb80, 0xffc0, "%sl.w %s",
0xebc0, 0xffc0, "%sl.l %s",
0xec00, 0xffc0, "unimplemented",
0xec40, 0xffc0, "%sr.b %s",
0xec80, 0xffc0, "%sr.w %s",

```


CHAPTER 10

```

/* A: 0xec80 */ SFTROT, 0, 0xec80, 0xffc0, "%sr.l %s",
/* A: 0xecc0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0xed00 */ SFTROT, 0, 0xed00, 0xffc0, "%sl.b %s",
/* A: 0xed40 */ SFTROT, 0, 0xed40, 0xffc0, "%sl.w %s",
/* A: 0xed80 */ SFTROT, 0, 0xed80, 0xffc0, "%sl.l %s",
/* A: 0xedc0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0xee00 */ SFTROT, 0, 0xee00, 0xffc0, "%sr.b %s",
/* A: 0xee40 */ SFTROT, 0, 0xee40, 0xffc0, "%sr.w %s",
/* A: 0xee80 */ SFTROT, 0, 0xee80, 0xffc0, "%sr.l %s",
/* A: 0xeec0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0xef00 */ SFTROT, 0, 0xef00, 0xffc0, "%sl.b %s",
/* A: 0xef40 */ SFTROT, 0, 0xef40, 0xffc0, "%sl.w %s",
/* A: 0xef80 */ SFTROT, 0, 0xef80, 0xffc0, "%sl.l %s",
/* A: 0xefc0 */ NONE, 0, 0x0000, 0x0000, unimplemented,
/* A: 0xf000 */ LINE_F, 0, 0xf000, 0xf000, "line F %s",
/* A: 0xf040 */ LINE_F, 0, 0xf040, 0xf000, "line F %s",
/* A: 0xf080 */ LINE_F, 0, 0xf080, 0xf000, "line F %s",
/* A: 0xf0c0 */ LINE_F, 0, 0xf0c0, 0xf000, "line F %s",
/* A: 0xf100 */ LINE_F, 0, 0xf100, 0xf000, "line F %s",
/* A: 0xf140 */ LINE_F, 0, 0xf140, 0xf000, "line F %s",
/* A: 0xf180 */ LINE_F, 0, 0xf180, 0xf000, "line F %s",
/* A: 0xf1c0 */ LINE_F, 0, 0xf1c0, 0xf000, "line F %s",
/* A: 0xf200 */ LINE_F, 0, 0xf200, 0xf000, "line F %s",
/* A: 0xf240 */ LINE_F, 0, 0xf240, 0xf000, "line F %s",
/* A: 0xf280 */ LINE_F, 0, 0xf280, 0xf000, "line F %s",
/* A: 0xf2c0 */ LINE_F, 0, 0xf2c0, 0xf000, "line F %s",
/* A: 0xf300 */ LINE_F, 0, 0xf300, 0xf000, "line F %s",
/* A: 0xf340 */ LINE_F, 0, 0xf340, 0xf000, "line F %s",
/* A: 0xf380 */ LINE_F, 0, 0xf380, 0xf000, "line F %s",
/* A: 0xf3c0 */ LINE_F, 0, 0xf3c0, 0xf000, "line F %s",
/* A: 0xf400 */ LINE_F, 0, 0xf400, 0xf000, "line F %s",
/* A: 0xf440 */ LINE_F, 0, 0xf440, 0xf000, "line F %s",
/* A: 0xf480 */ LINE_F, 0, 0xf480, 0xf000, "line F %s",
/* A: 0xf4c0 */ LINE_F, 0, 0xf4c0, 0xf000, "line F %s",
/* A: 0xf500 */ LINE_F, 0, 0xf500, 0xf000, "line F %s",
/* A: 0xf540 */ LINE_F, 0, 0xf540, 0xf000, "line F %s",
/* A: 0xf580 */ LINE_F, 0, 0xf580, 0xf000, "line F %s",
/* A: 0xf5c0 */ LINE_F, 0, 0xf5c0, 0xf000, "line F %s",
/* A: 0xf600 */ LINE_F, 0, 0xf600, 0xf000, "line F %s",
/* A: 0xf640 */ LINE_F, 0, 0xf640, 0xf000, "line F %s",
/* A: 0xf680 */ LINE_F, 0, 0xf680, 0xf000, "line F %s",
/* A: 0xf6c0 */ LINE_F, 0, 0xf6c0, 0xf000, "line F %s",
/* A: 0xf700 */ LINE_F, 0, 0xf700, 0xf000, "line F %s",
/* A: 0xf740 */ LINE_F, 0, 0xf740, 0xf000, "line F %s",
/* A: 0xf780 */ LINE_F, 0, 0xf780, 0xf000, "line F %s",
/* A: 0xf7c0 */ LINE_F, 0, 0xf7c0, 0xf000, "line F %s",
/* A: 0xf800 */ LINE_F, 0, 0xf800, 0xf000, "line F %s",
/* A: 0xf840 */ LINE_F, 0, 0xf840, 0xf000, "line F %s",
/* A: 0xf880 */ LINE_F, 0, 0xf880, 0xf000, "line F %s",
/* A: 0xf8c0 */ LINE_F, 0, 0xf8c0, 0xf000, "line F %s",
/* A: 0xf900 */ LINE_F, 0, 0xf900, 0xf000, "line F %s",
/* A: 0xf940 */ LINE_F, 0, 0xf940, 0xf000, "line F %s",
/* A: 0xf980 */ LINE_F, 0, 0xf980, 0xf000, "line F %s",
/* A: 0xf9c0 */ LINE_F, 0, 0xf9c0, 0xf000, "line F %s",
/* A: 0xfa00 */ LINE_F, 0, 0xfa00, 0xf000, "line F %s",
/* A: 0xfa40 */ LINE_F, 0, 0xfa40, 0xf000, "line F %s",
/* A: 0xfa80 */ LINE_F, 0, 0xfa80, 0xf000, "line F %s",
/* A: 0xfac0 */ LINE_F, 0, 0xfac0, 0xf000, "line F %s",
/* A: 0xfb00 */ LINE_F, 0, 0xfb00, 0xf000, "line F %s",
/* A: 0xfb40 */ LINE_F, 0, 0xfb40, 0xf000, "line F %s",
/* A: 0xfb80 */ LINE_F, 0, 0xfb80, 0xf000, "line F %s",
/* A: 0xfbc0 */ LINE_F, 0, 0xfbc0, 0xf000, "line F %s",
/* A: 0xfc00 */ LINE_F, 0, 0xfc00, 0xf000, "line F %s",
/* A: 0xfc40 */ LINE_F, 0, 0xfc40, 0xf000, "line F %s",
/* A: 0xfc80 */ LINE_F, 0, 0xfc80, 0xf000, "line F %s",
/* A: 0fcc0 */ LINE_F, 0, 0fcc0, 0xf000, "line F %s",
/* A: 0xfd00 */ LINE_F, 0, 0xfd00, 0xf000, "line F %s",

```



```

/* A: 0xfd40 */ LINE_F, 0,
/* A: 0xfd80 */ LINE_F, 0,
/* A: 0xfdc0 */ LINE_F, 0,
/* A: 0xfe00 */ LINE_F, 0,
/* A: 0xfe40 */ LINE_F, 0,
/* A: 0xfe80 */ LINE_F, 0,
/* A: 0xfec0 */ LINE_F, 0,
/* A: 0xff00 */ LINE_F, 0,
/* A: 0xff40 */ LINE_F, 0,
/* A: 0xff80 */ LINE_F, 0,
/* A: 0xffc0 */ LINE_F, 0,
    0, 0, 0,
};

```

```

struct hash_tab table_B[]
/* B: 0x1300 */ ONEREG, 0,
/* B: 0x0b80 */ LINKOP, 0,
/* B: 0x0d80 */ ONEREG, 0,
/* B: 0x0200 */ ONEREG, 0,
/* B: 0x0240 */ ONEREG, 0,
/* B: 0x0280 */ ONEREG, 0,
/* B: 0x02c0 */ ONEREG, 0,
/* B: 0x0300 */ ONEREG, 0,
/* B: 0x0340 */ ONEREG, 0,
/* B: 0x0380 */ ONEREG, 0,
/* B: 0x03c0 */ ONEREG, 0,
/* B: 0x0400 */ ONEREG, 0,
/* B: 0x0440 */ ONEREG, 0,
/* B: 0x0480 */ ONEREG, 0,
/* B: 0x04c0 */ ONEREG, 0,
/* B: 0x0500 */ ONEREG, 0,
/* B: 0x0540 */ ONEREG, 0,
/* B: 0x0580 */ ONEREG, 0,
/* B: 0x05c0 */ ONEREG, 0,
/* B: 0x0640 */ ONEREG, 0,
/* B: 0x0680 */ ONEREG, 0,
/* B: 0x06c0 */ ONEREG, 0,
/* B: 0x0700 */ ONEREG, 0,
/* B: 0x0740 */ ONEREG, 0,
/* B: 0x0780 */ ONEREG, 0,
/* B: 0x07c0 */ ONEREG, 0,
/* B: 0x0800 */ ONEREG, 0,
/* B: 0x0840 */ ONEREG, 0,
/* B: 0x0880 */ ONEREG, 0,
/* B: 0x08c0 */ ONEREG, 0,
/* B: 0x0900 */ ONEREG, 0,
/* B: 0x0940 */ ONEREG, 0,
/* B: 0x0980 */ ONEREG, 0,
/* B: 0x09c0 */ ONEREG, 0,
/* B: 0x0e00 */ ONEREG, 0,
/* B: 0x0e40 */ ONEREG, 0,
/* B: 0x0e80 */ ONEREG, 0,
/* B: 0x0ec0 */ ONEREG, 0,
/* B: 0x0f00 */ ONEREG, 0,
/* B: 0x0f40 */ ONEREG, 0,
/* B: 0x0f80 */ ONEREG, 0,
/* B: 0x0fc0 */ ONEREG, 0,
/* B: 0x1000 */ ONEREG, 0,
/* B: 0x1040 */ ONEREG, 0,
/* B: 0x1080 */ ONEREG, 0,
/* B: 0x10c0 */ ONEREG, 0,
/* B: 0x1100 */ ONEREG, 0,
/* B: 0x1140 */ ONEREG, 0,
/* B: 0x1180 */ ONEREG, 0,
/* B: 0x11c0 */ ONEREG, 0,
/* B: 0x1200 */ ONEREG, 0,
/* B: 0x1240 */ ONEREG, 0,
/* B: 0x1280 */ ONEREG, 0,

```

```

0xfd40, 0xf000, "line F %s",
0xfd80, 0xf000, "line F %s",
0xfdc0, 0xf000, "line F %s",
0xfe00, 0xf000, "line F %s",
0xfe40, 0xf000, "line F %s",
0xfe80, 0xf000, "line F %s",
0xfec0, 0xf000, "line F %s",
0xff00, 0xf000, "line F %s",
0xff40, 0xf000, "line F %s",
0xff80, 0xf000, "line F %s",
0xffc0, 0xf000, "line F %s",
0, 0

```

```

= (
0x4e75, 0xffff, "rts",
0x4e56, 0xffff, "link.w FP,%s",
0x4e5e, 0xffff, "unlk.w FP",
0x4880, 0xffff, "ext.w D0",
0x4881, 0xffff, "ext.w D1",
0x4882, 0xffff, "ext.w D2",
0x4883, 0xffff, "ext.w D3",
0x4884, 0xffff, "ext.w D4",
0x4885, 0xffff, "ext.w D5",
0x4886, 0xffff, "ext.w D6",
0x4887, 0xffff, "ext.w D7",
0x48c0, 0xffff, "ext.l D0",
0x48c1, 0xffff, "ext.l D1",
0x48c2, 0xffff, "ext.l D2",
0x48c3, 0xffff, "ext.l D3",
0x48c4, 0xffff, "ext.l D4",
0x48c5, 0xffff, "ext.l D5",
0x48c6, 0xffff, "ext.l D6",
0x48c7, 0xffff, "ext.l D7",
0x4e41, 0xffff, "trap #1",
0x4e42, 0xffff, "trap #2",
0x4e43, 0xffff, "trap #3",
0x4e44, 0xffff, "trap #4",
0x4e45, 0xffff, "trap #5",
0x4e46, 0xffff, "trap #6",
0x4e47, 0xffff, "trap #7",
0x4e48, 0xffff, "trap #8",
0x4e49, 0xffff, "trap #9",
0x4e4a, 0xffff, "trap #a",
0x4e4b, 0xffff, "trap #b",
0x4e4c, 0xffff, "trap #c",
0x4e4d, 0xffff, "trap #d",
0x4e4e, 0xffff, "trap #e",
0x4e4f, 0xffff, "trap #f",
0x4e60, 0xffff, "move.l A0,USP",
0x4e61, 0xffff, "move.l A1,USP",
0x4e62, 0xffff, "move.l A2,USP",
0x4e63, 0xffff, "move.l A3,USP",
0x4e64, 0xffff, "move.l A4,USP",
0x4e65, 0xffff, "move.l A5,USP",
0x4e66, 0xffff, "move.l FP,USP",
0x4e67, 0xffff, "move.l SP,USP",
0x4e68, 0xffff, "move.l USP,A0",
0x4e69, 0xffff, "move.l USP,A1",
0x4e6a, 0xffff, "move.l USP,A2",
0x4e6b, 0xffff, "move.l USP,A3",
0x4e6c, 0xffff, "move.l USP,A4",
0x4e6d, 0xffff, "move.l USP,A5",
0x4e6e, 0xffff, "move.l USP,FP",
0x4e6f, 0xffff, "move.l USP,SP",
0x4e70, 0xffff, "reset",
0x4e71, 0xffff, "nop",
0x4e72, 0xffff, "stop",

```

CHAPTER 10

```

/* B: 0x12c0 */ ONEREG, 0, 0x4e73, 0xffff, "rte",
/* B: 0x1340 */ ONEREG, 0, 0x4e76, 0xffff, "trapv",
/* B: 0x1380 */ ONEREG, 0, 0x4e77, 0xffff, "rtr",
/* B: 0x0000 */ ONEREG, 0, 0x4840, 0xffff, "swap.w D0",
/* B: 0x0040 */ ONEREG, 0, 0x4841, 0xffff, "swap.w D1",
/* B: 0x0080 */ ONEREG, 0, 0x4842, 0xffff, "swap.w D2",
/* B: 0x00c0 */ ONEREG, 0, 0x4843, 0xffff, "swap.w D3",
/* B: 0x0100 */ ONEREG, 0, 0x4844, 0xffff, "swap.w D4",
/* B: 0x0140 */ ONEREG, 0, 0x4845, 0xffff, "swap.w D5",
/* B: 0x0180 */ ONEREG, 0, 0x4846, 0xffff, "swap.w D6",
/* B: 0x01c0 */ ONEREG, 0, 0x4847, 0xffff, "swap.w D7",
/* B: 0x0a00 */ LINKOP, 0, 0x4e50, 0xffff, "link.w A0,%s",
/* B: 0x0a40 */ LINKOP, 0, 0x4e51, 0xffff, "link.w A1,%s",
/* B: 0x0a80 */ LINKOP, 0, 0x4e52, 0xffff, "link.w A2,%s",
/* B: 0x0ac0 */ LINKOP, 0, 0x4e53, 0xffff, "link.w A3,%s",
/* B: 0x0b00 */ LINKOP, 0, 0x4e54, 0xffff, "link.w A4,%s",
/* B: 0x0b40 */ LINKOP, 0, 0x4e55, 0xffff, "link.w A5,%s",
/* B: 0x0bc0 */ LINKOP, 0, 0x4e57, 0xffff, "link.w SP,%s",
/* B: 0x0c00 */ ONEREG, 0, 0x4e58, 0xffff, "unlk.w A0",
/* B: 0x0c40 */ ONEREG, 0, 0x4e59, 0xffff, "unlk.w A1",
/* B: 0x0c80 */ ONEREG, 0, 0x4e5a, 0xffff, "unlk.w A2",
/* B: 0x0cc0 */ ONEREG, 0, 0x4e5b, 0xffff, "unlk.w A3",
/* B: 0x0d00 */ ONEREG, 0, 0x4e5c, 0xffff, "unlk.w A4",
/* B: 0x0d40 */ ONEREG, 0, 0x4e5d, 0xffff, "unlk.w A5",
/* B: 0x0dc0 */ ONEREG, 0, 0x4e5f, 0xffff, "unlk.w SP",
/* B: 0x13c0 */ DBRNCH, 0, 0x50c8, 0xffff, "dbt D0,%s",
/* B: 0x1400 */ DBRNCH, 0, 0x51c8, 0xffff, "dbf D0,%s",
/* B: 0x1440 */ DBRNCH, 0, 0x52c8, 0xffff, "dbhi D0,%s",
/* B: 0x1480 */ DBRNCH, 0, 0x53c8, 0xffff, "dbls D0,%s",
/* B: 0x14c0 */ DBRNCH, 0, 0x54c8, 0xffff, "dbcc D0,%s",
/* B: 0x1500 */ DBRNCH, 0, 0x55c8, 0xffff, "dbcs D0,%s",
/* B: 0x1540 */ DBRNCH, 0, 0x56c8, 0xffff, "dbne D0,%s",
/* B: 0x1580 */ DBRNCH, 0, 0x57c8, 0xffff, "dbeq D0,%s",
/* B: 0x15c0 */ DBRNCH, 0, 0x58c8, 0xffff, "dbvc D0,%s",
/* B: 0x1600 */ DBRNCH, 0, 0x59c8, 0xffff, "dbvs D0,%s",
/* B: 0x1640 */ DBRNCH, 0, 0x5ac8, 0xffff, "dbpl D0,%s",
/* B: 0x1680 */ DBRNCH, 0, 0x5bc8, 0xffff, "dbmi D0,%s",
/* B: 0x16c0 */ DBRNCH, 0, 0x5cc8, 0xffff, "dbge D0,%s",
/* B: 0x1700 */ DBRNCH, 0, 0x5dc8, 0xffff, "dblt D0,%s",
/* B: 0x1740 */ DBRNCH, 0, 0x5ec8, 0xffff, "dbgt D0,%s",
/* B: 0x1780 */ DBRNCH, 0, 0x5fc8, 0xffff, "dblt D0,%s",
/* B: 0x17c0 */ DBRNCH, 0, 0x50c9, 0xffff, "dbt D1,%s",
/* B: 0x1800 */ DBRNCH, 0, 0x51c9, 0xffff, "dbf D1,%s",
/* B: 0x1840 */ DBRNCH, 0, 0x52c9, 0xffff, "dbhi D1,%s",
/* B: 0x1880 */ DBRNCH, 0, 0x53c9, 0xffff, "dbls D1,%s",
/* B: 0x18c0 */ DBRNCH, 0, 0x54c9, 0xffff, "dbcc D1,%s",
/* B: 0x1900 */ DBRNCH, 0, 0x55c9, 0xffff, "dbcs D1,%s",
/* B: 0x1940 */ DBRNCH, 0, 0x56c9, 0xffff, "dbne D1,%s",
/* B: 0x1980 */ DBRNCH, 0, 0x57c9, 0xffff, "dbeq D1,%s",
/* B: 0x19c0 */ DBRNCH, 0, 0x58c9, 0xffff, "dbvc D1,%s",
/* B: 0x1a00 */ DBRNCH, 0, 0x59c9, 0xffff, "dbvs D1,%s",
/* B: 0x1a40 */ DBRNCH, 0, 0x5ac9, 0xffff, "dbpl D1,%s",
/* B: 0x1a80 */ DBRNCH, 0, 0x5bc9, 0xffff, "dbmi D1,%s",
/* B: 0x1ac0 */ DBRNCH, 0, 0x5cc9, 0xffff, "dbge D1,%s",
/* B: 0x1b00 */ DBRNCH, 0, 0x5dc9, 0xffff, "dblt D1,%s",
/* B: 0x1b40 */ DBRNCH, 0, 0x5ec9, 0xffff, "dbgt D1,%s",
/* B: 0x1b80 */ DBRNCH, 0, 0x5fc9, 0xffff, "dblt D1,%s",
/* B: 0x1bc0 */ DBRNCH, 0, 0x50ca, 0xffff, "dbt D2,%s",
/* B: 0x1c00 */ DBRNCH, 0, 0x51ca, 0xffff, "dbf D2,%s",
/* B: 0x1c40 */ DBRNCH, 0, 0x52ca, 0xffff, "dbhi D2,%s",
/* B: 0x1c80 */ DBRNCH, 0, 0x53ca, 0xffff, "dbls D2,%s",
/* B: 0x1cc0 */ DBRNCH, 0, 0x54ca, 0xffff, "dbcc D2,%s",
/* B: 0x1d00 */ DBRNCH, 0, 0x55ca, 0xffff, "dbcs D2,%s",
/* B: 0x1d40 */ DBRNCH, 0, 0x56ca, 0xffff, "dbne D2,%s",
/* B: 0x1d80 */ DBRNCH, 0, 0x57ca, 0xffff, "dbeq D2,%s",
/* B: 0x1dc0 */ DBRNCH, 0, 0x58ca, 0xffff, "dbvc D2,%s",
/* B: 0x1e00 */ DBRNCH, 0, 0x59ca, 0xffff, "dbvs D2,%s",

```

```

/* B: 0x1e40 */ DBRNCH, 0, 0x5aca, 0xffff, "dbpl D2,%s",
/* B: 0x1e80 */ DBRNCH, 0, 0x5bca, 0xffff, "dbmi D2,%s",
/* B: 0x1ec0 */ DBRNCH, 0, 0x5cca, 0xffff, "dbge D2,%s",
/* B: 0x1f00 */ DBRNCH, 0, 0x5dca, 0xffff, "dblt D2,%s",
/* B: 0x1f40 */ DBRNCH, 0, 0x5eca, 0xffff, "dbgt D2,%s",
/* B: 0x1f80 */ DBRNCH, 0, 0x5fca, 0xffff, "db1e D2,%s",
/* B: 0x1fc0 */ DBRNCH, 0, 0x50cb, 0xffff, "dbt D3,%s",
/* B: 0x2000 */ DBRNCH, 0, 0x51cb, 0xffff, "dbf D3,%s",
/* B: 0x2040 */ DBRNCH, 0, 0x52cb, 0xffff, "dbhi D3,%s",
/* B: 0x2080 */ DBRNCH, 0, 0x53cb, 0xffff, "db1s D3,%s",
/* B: 0x20c0 */ DBRNCH, 0, 0x54cb, 0xffff, "dbcc D3,%s",
/* B: 0x2100 */ DBRNCH, 0, 0x55cb, 0xffff, "dbcs D3,%s",
/* B: 0x2140 */ DBRNCH, 0, 0x56cb, 0xffff, "dbne D3,%s",
/* B: 0x2180 */ DBRNCH, 0, 0x57cb, 0xffff, "dbeq D3,%s",
/* B: 0x21c0 */ DBRNCH, 0, 0x58cb, 0xffff, "dbvc D3,%s",
/* B: 0x2200 */ DBRNCH, 0, 0x59cb, 0xffff, "dbvs D3,%s",
/* B: 0x2240 */ DBRNCH, 0, 0x5acb, 0xffff, "dbpl D3,%s",
/* B: 0x2280 */ DBRNCH, 0, 0x5bcb, 0xffff, "dbmi D3,%s",
/* B: 0x22c0 */ DBRNCH, 0, 0x5ccb, 0xffff, "dbge D3,%s",
/* B: 0x2300 */ DBRNCH, 0, 0x5dcb, 0xffff, "dblt D3,%s",
/* B: 0x2340 */ DBRNCH, 0, 0x5ecb, 0xffff, "dbgt D3,%s",
/* B: 0x2380 */ DBRNCH, 0, 0x5fcb, 0xffff, "db1e D3,%s",
/* B: 0x23c0 */ DBRNCH, 0, 0x50cc, 0xffff, "dbt D4,%s",
/* B: 0x2400 */ DBRNCH, 0, 0x51cc, 0xffff, "dbf D4,%s",
/* B: 0x2440 */ DBRNCH, 0, 0x52cc, 0xffff, "dbhi D4,%s",
/* B: 0x2480 */ DBRNCH, 0, 0x53cc, 0xffff, "db1s D4,%s",
/* B: 0x24c0 */ DBRNCH, 0, 0x54cc, 0xffff, "dbcc D4,%s",
/* B: 0x2500 */ DBRNCH, 0, 0x55cc, 0xffff, "dbcs D4,%s",
/* B: 0x2540 */ DBRNCH, 0, 0x56cc, 0xffff, "dbne D4,%s",
/* B: 0x2580 */ DBRNCH, 0, 0x57cc, 0xffff, "dbeq D4,%s",
/* B: 0x25c0 */ DBRNCH, 0, 0x58cc, 0xffff, "dbvc D4,%s",
/* B: 0x2600 */ DBRNCH, 0, 0x59cc, 0xffff, "dbvs D4,%s",
/* B: 0x2640 */ DBRNCH, 0, 0x5acc, 0xffff, "dbpl D4,%s",
/* B: 0x2680 */ DBRNCH, 0, 0x5bcc, 0xffff, "dbmi D4,%s",
/* B: 0x26c0 */ DBRNCH, 0, 0x5ccc, 0xffff, "dbge D4,%s",
/* B: 0x2700 */ DBRNCH, 0, 0x5dcc, 0xffff, "dblt D4,%s",
/* B: 0x2740 */ DBRNCH, 0, 0x5ecc, 0xffff, "dbgt D4,%s",
/* B: 0x2780 */ DBRNCH, 0, 0x5fcc, 0xffff, "db1e D4,%s",
/* B: 0x27c0 */ DBRNCH, 0, 0x50cd, 0xffff, "dbt D5,%s",
/* B: 0x2800 */ DBRNCH, 0, 0x51cd, 0xffff, "dbf D5,%s",
/* B: 0x2840 */ DBRNCH, 0, 0x52cd, 0xffff, "dbhi D5,%s",
/* B: 0x2880 */ DBRNCH, 0, 0x53cd, 0xffff, "db1s D5,%s",
/* B: 0x28c0 */ DBRNCH, 0, 0x54cd, 0xffff, "dbcc D5,%s",
/* B: 0x2900 */ DBRNCH, 0, 0x55cd, 0xffff, "dbcs D5,%s",
/* B: 0x2940 */ DBRNCH, 0, 0x56cd, 0xffff, "dbne D5,%s",
/* B: 0x2980 */ DBRNCH, 0, 0x57cd, 0xffff, "dbeq D5,%s",
/* B: 0x29c0 */ DBRNCH, 0, 0x58cd, 0xffff, "dbvc D5,%s",
/* B: 0x2a00 */ DBRNCH, 0, 0x59cd, 0xffff, "dbvs D5,%s",
/* B: 0x2a40 */ DBRNCH, 0, 0x5acd, 0xffff, "dbpl D5,%s",
/* B: 0x2a80 */ DBRNCH, 0, 0x5bcd, 0xffff, "dbmi D5,%s",
/* B: 0x2ac0 */ DBRNCH, 0, 0x5ccd, 0xffff, "dbge D5,%s",
/* B: 0x2b00 */ DBRNCH, 0, 0x5dcd, 0xffff, "dblt D5,%s",
/* B: 0x2b40 */ DBRNCH, 0, 0x5ecd, 0xffff, "dbgt D5,%s",
/* B: 0x2b80 */ DBRNCH, 0, 0x5fcd, 0xffff, "db1e D5,%s",
/* B: 0x2bc0 */ DBRNCH, 0, 0x50ce, 0xffff, "dbt D6,%s",
/* B: 0x2c00 */ DBRNCH, 0, 0x51ce, 0xffff, "dbf D6,%s",
/* B: 0x2c40 */ DBRNCH, 0, 0x52ce, 0xffff, "dbhi D6,%s",
/* B: 0x2c80 */ DBRNCH, 0, 0x53ce, 0xffff, "db1s D6,%s",
/* B: 0x2cc0 */ DBRNCH, 0, 0x54ce, 0xffff, "dbcc D6,%s",
/* B: 0x2d00 */ DBRNCH, 0, 0x55ce, 0xffff, "dbcs D6,%s",
/* B: 0x2d40 */ DBRNCH, 0, 0x56ce, 0xffff, "dbne D6,%s",
/* B: 0x2d80 */ DBRNCH, 0, 0x57ce, 0xffff, "dbeq D6,%s",
/* B: 0x2dc0 */ DBRNCH, 0, 0x58ce, 0xffff, "dbvc D6,%s",
/* B: 0x2e00 */ DBRNCH, 0, 0x59ce, 0xffff, "dbvs D6,%s",
/* B: 0x2e40 */ DBRNCH, 0, 0x5ace, 0xffff, "dbpl D6,%s",
/* B: 0x2e80 */ DBRNCH, 0, 0x5bce, 0xffff, "dbmi D6,%s",

```


CHAPTER 10

```

/* B: 0x2ec0 */ DBRNCH, 0, 0x5cce, 0xffff, "dbge D6,%s",
/* B: 0x2f00 */ DBRNCH, 0, 0x5dce, 0xffff, "dblt D6,%s",
/* B: 0x2f40 */ DBRNCH, 0, 0x5ece, 0xffff, "dbgt D6,%s",
/* B: 0x2f80 */ DBRNCH, 0, 0x5fce, 0xffff, "dblt D6,%s",
/* B: 0x2fc0 */ DBRNCH, 0, 0x50cf, 0xffff, "dbt D7,%s",
/* B: 0x3000 */ DBRNCH, 0, 0x51cf, 0xffff, "dbf D7,%s",
/* B: 0x3040 */ DBRNCH, 0, 0x52cf, 0xffff, "dbhi D7,%s",
/* B: 0x3080 */ DBRNCH, 0, 0x53cf, 0xffff, "dbls D7,%s",
/* B: 0x30c0 */ DBRNCH, 0, 0x54cf, 0xffff, "dbcc D7,%s",
/* B: 0x3100 */ DBRNCH, 0, 0x55cf, 0xffff, "dbcs D7,%s",
/* B: 0x3140 */ DBRNCH, 0, 0x56cf, 0xffff, "dbne D7,%s",
/* B: 0x3180 */ DBRNCH, 0, 0x57cf, 0xffff, "dbeq D7,%s",
/* B: 0x31c0 */ DBRNCH, 0, 0x58cf, 0xffff, "dbvc D7,%s",
/* B: 0x3200 */ DBRNCH, 0, 0x59cf, 0xffff, "dbvs D7,%s",
/* B: 0x3240 */ DBRNCH, 0, 0x5acf, 0xffff, "dbpl D7,%s",
/* B: 0x3280 */ DBRNCH, 0, 0x5bcf, 0xffff, "dbmi D7,%s",
/* B: 0x32c0 */ DBRNCH, 0, 0x5ccf, 0xffff, "dbge D7,%s",
/* B: 0x3300 */ DBRNCH, 0, 0x5dcf, 0xffff, "dblt D7,%s",
/* B: 0x3340 */ DBRNCH, 0, 0x5ecf, 0xffff, "dbgt D7,%s",
/* B: 0x3380 */ DBRNCH, 0, 0x5fcf, 0xffff, "dblt D7,%s",
/* B: 0x33c0 */ BCDREG, 0, 0x8100, 0xffff, "sbcd %s",
/* B: 0x3400 */ BCDREG, 0, 0x8300, 0xffff, "sbcd %s",
/* B: 0x3440 */ BCDREG, 0, 0x8500, 0xffff, "sbcd %s",
/* B: 0x3480 */ BCDREG, 0, 0x8700, 0xffff, "sbcd %s",
/* B: 0x34c0 */ BCDREG, 0, 0x8900, 0xffff, "sbcd %s",
/* B: 0x3500 */ BCDREG, 0, 0x8b00, 0xffff, "sbcd %s",
/* B: 0x3540 */ BCDREG, 0, 0x8d00, 0xffff, "sbcd %s",
/* B: 0x3580 */ BCDREG, 0, 0x8f00, 0xffff, "sbcd %s",
/* B: 0x35c0 */ BCDREG, 0, 0x9100, 0xffff, "subx.b %s",
/* B: 0x3600 */ BCDREG, 0, 0x9300, 0xffff, "subx.b %s",
/* B: 0x3640 */ BCDREG, 0, 0x9500, 0xffff, "subx.b %s",
/* B: 0x3680 */ BCDREG, 0, 0x9700, 0xffff, "subx.b %s",
/* B: 0x36c0 */ BCDREG, 0, 0x9900, 0xffff, "subx.b %s",
/* B: 0x3700 */ BCDREG, 0, 0x9b00, 0xffff, "subx.b %s",
/* B: 0x3740 */ BCDREG, 0, 0x9d00, 0xffff, "subx.b %s",
/* B: 0x3780 */ BCDREG, 0, 0x9f00, 0xffff, "subx.b %s",
/* B: 0x37c0 */ BCDREG, 0, 0x9140, 0xffff, "subx.w %s",
/* B: 0x3800 */ BCDREG, 0, 0x9340, 0xffff, "subx.w %s",
/* B: 0x3840 */ BCDREG, 0, 0x9540, 0xffff, "subx.w %s",
/* B: 0x3880 */ BCDREG, 0, 0x9740, 0xffff, "subx.w %s",
/* B: 0x38c0 */ BCDREG, 0, 0x9940, 0xffff, "subx.w %s",
/* B: 0x3900 */ BCDREG, 0, 0x9b40, 0xffff, "subx.w %s",
/* B: 0x3940 */ BCDREG, 0, 0x9d40, 0xffff, "subx.w %s",
/* B: 0x3980 */ BCDREG, 0, 0x9f40, 0xffff, "subx.w %s",
/* B: 0x39c0 */ BCDREG, 0, 0x9180, 0xffff, "subx.l %s",
/* B: 0x3a00 */ BCDREG, 0, 0x9380, 0xffff, "subx.l %s",
/* B: 0x3a40 */ BCDREG, 0, 0x9580, 0xffff, "subx.l %s",
/* B: 0x3a80 */ BCDREG, 0, 0x9780, 0xffff, "subx.l %s",
/* B: 0x3ac0 */ BCDREG, 0, 0x9980, 0xffff, "subx.l %s",
/* B: 0x3b00 */ BCDREG, 0, 0x9b80, 0xffff, "subx.l %s",
/* B: 0x3b40 */ BCDREG, 0, 0x9d80, 0xffff, "subx.l %s",
/* B: 0x3b80 */ BCDREG, 0, 0x9f80, 0xffff, "subx.l %s",
/* B: 0x3bc0 */ CMPREG, 0, 0xb108, 0xffff, "cmpm.b A0,%s",
/* B: 0x3c00 */ CMPREG, 0, 0xb308, 0xffff, "cmpm.b A1,%s",
/* B: 0x3c40 */ CMPREG, 0, 0xb508, 0xffff, "cmpm.b A2,%s",
/* B: 0x3c80 */ CMPREG, 0, 0xb708, 0xffff, "cmpm.b A3,%s",
/* B: 0x3cc0 */ CMPREG, 0, 0xb908, 0xffff, "cmpm.b A4,%s",
/* B: 0x3d00 */ CMPREG, 0, 0xbb08, 0xffff, "cmpm.b A5,%s",
/* B: 0x3d40 */ CMPREG, 0, 0xbd08, 0xffff, "cmpm.b SP,%s",
/* B: 0x3d80 */ CMPREG, 0, 0xbf08, 0xffff, "cmpm.b FP,%s",
/* B: 0x3dc0 */ CMPREG, 0, 0xb148, 0xffff, "cmpm.w A0,%s",
/* B: 0x3e00 */ CMPREG, 0, 0xb348, 0xffff, "cmpm.w A1,%s",
/* B: 0x3e40 */ CMPREG, 0, 0xb548, 0xffff, "cmpm.w A2,%s",
/* B: 0x3e80 */ CMPREG, 0, 0xb748, 0xffff, "cmpm.w A3,%s",
/* B: 0x3ec0 */ CMPREG, 0, 0xb948, 0xffff, "cmpm.w A4,%s",
/* B: 0x3f00 */ CMPREG, 0, 0xbb48, 0xffff, "cmpm.w A5,%s",
/* B: 0x3f40 */ CMPREG, 0, 0xbd48, 0xffff, "cmpm.w FP,%s",

```



```

/* B: 0x3f80 */ CMPREG, 0,
/* B: 0x3fc0 */ CMPREG, 0,
/* B: 0x4000 */ CMPREG, 0,
/* B: 0x4040 */ CMPREG, 0,
/* B: 0x4080 */ CMPREG, 0,
/* B: 0x40c0 */ CMPREG, 0,
/* B: 0x4100 */ CMPREG, 0,
/* B: 0x4140 */ CMPREG, 0,
/* B: 0x4180 */ CMPREG, 0,
/* B: 0x41c0 */ EXDREG, 0,
/* B: 0x4200 */ EXDREG, 0,
/* B: 0x4240 */ EXDREG, 0,
/* B: 0x4280 */ EXDREG, 0,
/* B: 0x42c0 */ EXDREG, 0,
/* B: 0x4300 */ EXDREG, 0,
/* B: 0x4340 */ EXDREG, 0,
/* B: 0x4380 */ EXDREG, 0,
/* B: 0x43c0 */ EXAREG, 0,
/* B: 0x4400 */ EXAREG, 0,
/* B: 0x4440 */ EXAREG, 0,
/* B: 0x4480 */ EXAREG, 0,
/* B: 0x44c0 */ EXAREG, 0,
/* B: 0x4500 */ EXAREG, 0,
/* B: 0x4540 */ EXAREG, 0,
/* B: 0x4580 */ EXAREG, 0,
/* B: 0x45c0 */ EXAREG, 0,
/* B: 0x4600 */ EXAREG, 0,
/* B: 0x4640 */ EXAREG, 0,
/* B: 0x4680 */ EXAREG, 0,
/* B: 0x46c0 */ EXAREG, 0,
/* B: 0x4700 */ EXAREG, 0,
/* B: 0x4740 */ EXAREG, 0,
/* B: 0x4780 */ EXAREG, 0,
/* B: 0x47c0 */ BCDREG, 0,
/* B: 0x4800 */ BCDREG, 0,
/* B: 0x4840 */ BCDREG, 0,
/* B: 0x4880 */ BCDREG, 0,
/* B: 0x48c0 */ BCDREG, 0,
/* B: 0x4900 */ BCDREG, 0,
/* B: 0x4940 */ BCDREG, 0,
/* B: 0x4980 */ BCDREG, 0,
/* B: 0x49c0 */ BCDREG, 0,
/* B: 0x4a00 */ BCDREG, 0,
/* B: 0x4a40 */ BCDREG, 0,
/* B: 0x4a80 */ BCDREG, 0,
/* B: 0x4ac0 */ BCDREG, 0,
/* B: 0x4b00 */ BCDREG, 0,
/* B: 0x4b40 */ BCDREG, 0,
/* B: 0x4b80 */ BCDREG, 0,
/* B: 0x4bc0 */ BCDREG, 0,
/* B: 0x4c00 */ BCDREG, 0,
/* B: 0x4c40 */ BCDREG, 0,
/* B: 0x4c80 */ BCDREG, 0,
/* B: 0x4cc0 */ BCDREG, 0,
/* B: 0x4d00 */ BCDREG, 0,
/* B: 0x4d40 */ BCDREG, 0,
/* B: 0x4d80 */ BCDREG, 0,
/* B: 0x4dc0 */ BCDREG, 0,
/* B: 0x4e00 */ BCDREG, 0,
/* B: 0x4e40 */ BCDREG, 0,
/* B: 0x4e80 */ BCDREG, 0,
/* B: 0x4ec0 */ BCDREG, 0,
/* B: 0x4f00 */ BCDREG, 0,
/* B: 0x4f40 */ BCDREG, 0,
/* B: 0x4f80 */ BCDREG, 0,
/* B: 0x4fc0 */ MOVE_P, 0,
0xbf48, 0xffff, "cmpm.w SP,%s",
0xb188, 0xffff, "cmpm.l A0,%s",
0xb388, 0xffff, "cmpm.l A1,%s",
0xb588, 0xffff, "cmpm.l A2,%s",
0xb788, 0xffff, "cmpm.l A3,%s",
0xb988, 0xffff, "cmpm.l A4,%s",
0xbb88, 0xffff, "cmpm.l A5,%s",
0xbd88, 0xffff, "cmpm.l FP,%s",
0xbf88, 0xffff, "cmpm.l SP,%s",
0xc140, 0xffff, "exg.l D0,%s",
0xc340, 0xffff, "exg.l D1,%s",
0xc540, 0xffff, "exg.l D2,%s",
0xc740, 0xffff, "exg.l D3,%s",
0xc940, 0xffff, "exg.l D4,%s",
0xcb40, 0xffff, "exg.l D5,%s",
0xcd40, 0xffff, "exg.l D6,%s",
0xcf40, 0xffff, "exg.l D7,%s",
0xc148, 0xffff, "exg.l A0,%s",
0xc348, 0xffff, "exg.l A1,%s",
0xc548, 0xffff, "exg.l A2,%s",
0xc748, 0xffff, "exg.l A3,%s",
0xc948, 0xffff, "exg.l A4,%s",
0xcb48, 0xffff, "exg.l A5,%s",
0xcd48, 0xffff, "exg.l FP,%s",
0xcf48, 0xffff, "exg.l SP,%s",
0xc188, 0xffff, "exg.l D0,%s",
0xc388, 0xffff, "exg.l D1,%s",
0xc588, 0xffff, "exg.l D2,%s",
0xc788, 0xffff, "exg.l D3,%s",
0xc988, 0xffff, "exg.l D4,%s",
0xcb88, 0xffff, "exg.l D5,%s",
0xcd88, 0xffff, "exg.l D6,%s",
0xcf88, 0xffff, "exg.l D7,%s",
0xc100, 0xffff, "abcd %s",
0xc300, 0xffff, "abcd %s",
0xc500, 0xffff, "abcd %s",
0xc700, 0xffff, "abcd %s",
0xc900, 0xffff, "abcd %s",
0xcb00, 0xffff, "abcd %s",
0xcd00, 0xffff, "abcd %s",
0xcf00, 0xffff, "abcd %s",
0xd100, 0xffff, "addx.b %s",
0xd300, 0xffff, "addx.b %s",
0xd500, 0xffff, "addx.b %s",
0xd700, 0xffff, "addx.b %s",
0xd900, 0xffff, "addx.b %s",
0xdb00, 0xffff, "addx.b %s",
0xdd00, 0xffff, "addx.b %s",
0xdf00, 0xffff, "addx.b %s",
0xd140, 0xffff, "addx.w %s",
0xd340, 0xffff, "addx.w %s",
0xd540, 0xffff, "addx.w %s",
0xd740, 0xffff, "addx.w %s",
0xd940, 0xffff, "addx.w %s",
0xdb40, 0xffff, "addx.w %s",
0xdd40, 0xffff, "addx.w %s",
0xdf40, 0xffff, "addx.w %s",
0xd180, 0xffff, "addx.l %s",
0xd380, 0xffff, "addx.l %s",
0xd580, 0xffff, "addx.l %s",
0xd780, 0xffff, "addx.l %s",
0xd980, 0xffff, "addx.l %s",
0xdb80, 0xffff, "addx.l %s",
0xdd80, 0xffff, "addx.l %s",
0xdf80, 0xffff, "addx.l %s",
0xd100, 0xffff, "movp.w %s,D0",

```

```

/* B: 0x5000 */ MOVE_P, 0, 0x0140, 0xffff, "movp.l %s,D0",
/* B: 0x5040 */ MOVE_P, 0, 0x0180, 0xffff, "movp.w D0,%s",
/* B: 0x5080 */ MOVE_P, 0, 0x01c0, 0xffff, "movp.l D0,%s",
/* B: 0x50c0 */ MOVE_P, 0, 0x0300, 0xffff, "movp.w %s,D1",
/* B: 0x5100 */ MOVE_P, 0, 0x0340, 0xffff, "movp.l %s,D1",
/* B: 0x5140 */ MOVE_P, 0, 0x0380, 0xffff, "movp.w D1,%s",
/* B: 0x5180 */ MOVE_P, 0, 0x03c0, 0xffff, "movp.l D1,%s",
/* B: 0x51c0 */ MOVE_P, 0, 0x0500, 0xffff, "movp.w %s,D2",
/* B: 0x5200 */ MOVE_P, 0, 0x0540, 0xffff, "movp.l %s,D2",
/* B: 0x5240 */ MOVE_P, 0, 0x0580, 0xffff, "movp.w D2,%s",
/* B: 0x5280 */ MOVE_P, 0, 0x05c0, 0xffff, "movp.l D2,%s",
/* B: 0x52c0 */ MOVE_P, 0, 0x0700, 0xffff, "movp.w %s,D3",
/* B: 0x5300 */ MOVE_P, 0, 0x0740, 0xffff, "movp.l %s,D3",
/* B: 0x5340 */ MOVE_P, 0, 0x0780, 0xffff, "movp.w D3,%s",
/* B: 0x5380 */ MOVE_P, 0, 0x07c0, 0xffff, "movp.l D3,%s",
/* B: 0x53c0 */ MOVE_P, 0, 0x0900, 0xffff, "movp.w %s,D4",
/* B: 0x5400 */ MOVE_P, 0, 0x0940, 0xffff, "movp.l %s,D4",
/* B: 0x5440 */ MOVE_P, 0, 0x0980, 0xffff, "movp.w D4,%s",
/* B: 0x5480 */ MOVE_P, 0, 0x09c0, 0xffff, "movp.l D4,%s",
/* B: 0x54c0 */ MOVE_P, 0, 0x0b00, 0xffff, "movp.w %s,D5",
/* B: 0x5500 */ MOVE_P, 0, 0x0b40, 0xffff, "movp.l %s,D5",
/* B: 0x5540 */ MOVE_P, 0, 0x0b80, 0xffff, "movp.w D5,%s",
/* B: 0x5580 */ MOVE_P, 0, 0x0bc0, 0xffff, "movp.l D5,%s",
/* B: 0x55c0 */ MOVE_P, 0, 0x0d00, 0xffff, "movp.w %s,D6",
/* B: 0x5600 */ MOVE_P, 0, 0x0d40, 0xffff, "movp.l %s,D6",
/* B: 0x5640 */ MOVE_P, 0, 0x0d80, 0xffff, "movp.w D6,%s",
/* B: 0x5680 */ MOVE_P, 0, 0x0dc0, 0xffff, "movp.l D6,%s",
/* B: 0x56c0 */ MOVE_P, 0, 0x0f00, 0xffff, "movp.w %s,D7",
/* B: 0x5700 */ MOVE_P, 0, 0x0f40, 0xffff, "movp.l %s,D7",
/* B: 0x5740 */ MOVE_P, 0, 0x0f80, 0xffff, "movp.w D7,%s",
/* B: 0x5780 */ MOVE_P, 0, 0x0fc0, 0xffff, "movp.l D7,%s",
/* B: 0x57c0 */ IMCCR, 0, 0x023c, 0xffff, "andi.b %s,CCR",
/* B: 0x5800 */ IMMSR, 0, 0x027c, 0xffff, "and.w %s,SR",
/* B: 0x5840 */ IMCCR, 0, 0x0a3c, 0xffff, "eori.b %s,CCR",
/* B: 0x5880 */ IMMSR, 0, 0x0a7c, 0xffff, "eori.w %s,SR",
/* B: 0x0600 */ ONEREG, 0, 0x4afc, 0xffff, "illegal",
    0, 0, 0, 0
};

```

The linkacc.bat and linkacc.arg files

The linkacc.bat and linkacc.arg files (Programs 10-10 and 10-11) are used to link the accessory. They are similar to the versions in the previous chapter, except they contain the new functions.

Program 10-10. linkacc.bat

```

c:\bin\link68 [undefined,symbols,command[linkacc.arg]]
c:\bin\relmod a
c:\bin\rn a.68k
c:\bin\wait

```

Program 10-11. linkacc.arg

```
a.68k=c:accstart.o,main.o,
```

```

CONFIGAC.O,BT.O,DONEWND.O,ISMATCH.O,GETBASE.O,DECSTAT.O,GETNAME.O,
SHOWWND.O,GETSYMS.O,TRACE.O,OPENDATA.O,ERRORS.O,SETTOP.O,GETLONG.O,
BOMBINFO.O,GETREAL.O,GETARGS.O,

```

```
GIVEHELP.O,GOTKEY.O,DOIT.O,GETTRACE.O,
```

CHARTER SUBSCRIPTION FORM

☐ Payment enclosed ☐ Charge my VISA/MasterCard

☐ **YES!**

Sign me up for six issues (a full year's subscription) at the special introductory price of just \$59.95. I save more than \$17 off the newsstand price.

Credit Card # _____ Exp. Date _____

Signature _____

Name _____

Address _____

City _____ State _____ Zip _____

Outside U.S.A., please add \$6 (U.S.) per year for postage.

**CLIP THIS
AND SAVE \$17**

Here's your chance to cash in with big savings on *COMPUTE!'s Atari ST Disk & Magazine*—the exciting new publication devoted exclusively to the special needs and interests of Atari ST users like you.

Every other month, *COMPUTE!'s Atari ST Disk & Magazine* brings you exciting new action-packed programs already on disk! Just load and you're ready to run.

You can depend on getting at least five new programs in each issue—high-quality applications, educational, home finance, utility, and game programs you and the entire family will use, enjoy, and profit from all year long.

And here's even more good news. Subscribe now to *COMPUTE!'s Atari ST Disk & Magazine* and take advantage of big Charter Subscription savings. Get a full year's subscription for just \$59.95. You save over \$17 off the newsstand price.

No other publication gives you more for your Atari ST than *COMPUTE!'s Atari ST Disk & Magazine*. So sign up now by using the coupon above—or call 1-800-247-5470 (in Iowa 1-800-532-1272).



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7551 DES MOINES, IA

POSTAGE WILL BE PAID BY ADDRESSEE

COMPUTE!'s Atari ST Disk & Magazine

P.O. Box 10775

Des Moines, IA 50347-0775



NEW FOR ATARI ST USERS

COMPUTE!'s ATARI ST DISK & MAGAZINE

Only COMPUTE!'s Atari ST Disk & Magazine gives you all this and more in each big issue:

TOP QUALITY PROGRAMS: Application programs for home and business. Utilities. Games. Educational programs for the youngsters. All are already on an enclosed disk and ready to run. For example: a typical disk might contain an elaborate adventure game written in BASIC, a programming utility written in machine language, a dazzling graphics demo in compiled Pascal, and a useful home or business application written in Forth or C.

NEOCHROME OF THE MONTH: What are computer artists doing with the Atari ST? Each issue contains a Neochrome picture file—ready to load and admire.

REGULAR COLUMNS: If you're a programmer—or would like to be—you'll love our col-

umns on ST programming techniques and the C language. Or check out our column on the latest events and happenings throughout the ST community. Or send your questions and helpful hints to our Reader's Feedback column.

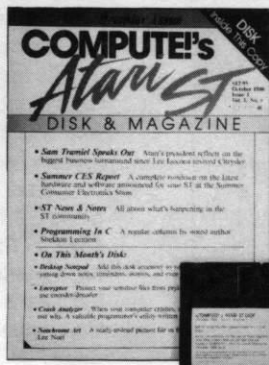
REVIEWS: Honest evaluations of the latest, best software and hardware for the Atari ST.

NEWS & PRODUCTS: A comprehensive listing of all the new software and peripherals for your ST.

AND MORE: Interviews with ST newsmakers, reports on the latest industry trade shows, and overviews of significant new product introductions.

Don't miss a single big issue. Subscribe to **COMPUTE!'s Atari ST Disk & Magazine** now through this special money-saving offer. Return coupon above or call 1-800-247-5470 (in Iowa 1-800-532-1272).

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



RETURN COUPON ABOVE TO ENJOY
CHARTER SUBSCRIPTION PRIVILEGES


```
DISASSEM.O, SETUPDIS.O, IMMEDIAT.O, GETDIS.O, PCABSIMM.O, MATCHB.O,
TABLES.O, EFFADD.O, ADDRMODE.O,
```

```
accsup.o, env.a, vdibind, vdidata.o, gemlib, aesbind, osbind, libf
```

The test Program

Finally, there is a simple program that tests the accessory by allowing the user to deliberately crash a program in any of the seven ways that have been discussed in these two chapters. This is a separate program, and does not get linked in with the desk accessory.

The test program arranges for a nice stack trace by nesting several function calls before getting down to business. Each function has a distinctive set of arguments, to show how the stack trace works. The bomb routine is the subroutine that prints a menu and asks for a number to select which type of error the user would like to cause. When the user selects a number, the appropriate routine is called, and it prints out the address just before the instruction that will cause the crash, to confirm that the debugging aid is getting the right information.

Some of the routines that cause the crashes need special help from assembly language, since there is no way to cause TRAPV, CHK, ILLEGAL, or RESET instructions from C. In addition, the trick used to find the address of the errant instruction involves using an asm function to put a label just before the instruction, which C can reference and print.

Since TOS ignores division by zero errors, the zerodiv routine will return, unlike any of the others. The message "Oops! didn't die!!!!" is printed, and the program waits for a key to be pressed before terminating normally. The debugging aid will still have caught the exception, and the trace and disassembly listings will reflect the division error. The bombinfo printout will not reflect the divide error, since it relies on information TOS saves after the error, and TOS ignored it.

Program 10-12. test.c

```
main(ac,av)
int ac;
char **av;{

    int main_arg    = 0;

    foo(main_arg,1,2,3,4,5);
}
foo(main_arg,a,b,c,d,e)
int main_arg, a, b, c, d, e;{

    int (*func_ptr)();
    int bar();
```

CHAPTER 10

```

func_ptr = bar;
(*func_ptr)(6,7,8);
}
bar(x,y,z)
int x, y, z;{

    int a = 12;

    foozle(9,10,11,a);
}
foozle(x, y, z, a)
int x, y, z, a;{

    int f = 'f';

    bomb('a','b','c','d','e',f);
}
bomb(a,b,c,d,e,f)
int a, b, c, d, e, f;{

    int x;

    for( x = 0; x < 25; x++ )
        printf("\n");
    printf("Enter 1 for bus error\n");
    printf("Enter 2 for address error\n");
    printf("Enter 3 for illegal instruction\n");
    printf("Enter 4 for zero divide\n");
    printf("Enter 5 for CHK instruction\n");
    printf("Enter 6 for TRAPV instruction\n");
    printf("Enter 7 for Privilege violation\n");
    switch(bios(2,2)){
        case '1': bus_error(); break;
        case '2': addr_error(); break;
        case '3': illegal(); break;
        case '4': zerodiv(); break;
        case '5': chk_instr(); break;
        case '6': trapv_instr(); break;
        case '7': priv_error(); break;
        default: printf("No error generated\n"); break;
    }
    printf("Oops! didn't die!!!!\n");
    bios(2,2);
}
bus_error(){

    extern dieloc1();
    char *ptr = 0x0L;

    printf("About to bus error near location %X\n",dieloc1);
    asm("_dieloc1:");
    *ptr = 0;
}
addr_error(){

    extern dieloc2();
    int *ptr = 0x1L;

    printf("About to address error near location %X\n",dieloc2);
    asm("_dieloc2:");
    *ptr = 0;
}
illegal(){

    extern dieloc3();

    printf("About to bomb near location %X\n",dieloc3);
}

```

```

asm("_dieloc3:");
asm("illegal");
}
zerodiv(){
    extern dieloc4();
    int x, y;

    x = 5;
    y = 0;
    printf("About to divide by zero near location %X\n",dieloc4);
asm("_dieloc4:");
    return( x / y );
}
chk_instr(){
    extern dieloc5();

    printf("About to bomb near location %X\n",dieloc5);
asm("_dieloc5:");
asm("move.l #4,R0");
asm("chk.w #3,R0");
}
trapv_instr(){
    extern dieloc6();

    printf("About to bomb near location %X\n",dieloc6);
asm("_dieloc6:");
asm("move.w #2,CCR");
asm("trapv");
}
priv_error(){
    extern dieloc7();

    printf("About to bomb near location %X\n",dieloc7);
asm("_dieloc7:");
asm("reset");
}

```

The linktst.bat File

To link the test program, use the linktst.bat file, Program 10-13.

Program 10-13. linktst.bat

```

c:\bin\link68 [u,s] a.68k=c:gemstart.o,TEST.O,vdibind,gemlib,aesbind,osbind,libf
c:\bin\relmod a
c:\bin\rn a.68k
c:\bin\wait

```




Appendix

World Map Data

■ This appendix includes the data for World Map (world.c), discussed in Chapter 3. The Program listed in this appendix must be used with the discussion and program described in Chapter 3.

If you decide to type the data in instead of purchasing the disk with the data, you may want to draw only certain coastlines to save time (and finger fatigue, not to mention boredom). You can type in any part of the data you wish; just be sure you type in an entire segment. Each segment begins with a -1 and continues up to, but not including, the next -1. Type in as many segments as you wish. As you look through the data segments you'll notice that they have been printed in order of size, starting with the largest coastlines. Remember, though, that the more segments you type in, the more of the world will appear on the screen.

Program A-1. world.c

```
/*  
** Map coordinates are stored as row * FUDGEY + col  
**
```

```
long int world[] = {  
    -1,128177,128817,129456,130737,131376,132015,132656,133937,  
    134578,135220,136500,137142,138422,139704,140985,141626,141627,  
    142269,142910,142912,143553,144195,145475,146755,147395,148674,  
    149314,150594,151234,152514,153793,154433,155713,156352,157633,  
    158912,159552,160831,162110,162750,164029,164669,165949,167229,  
    167869,168510,168511,169151,169790,171070,171710,172349,172989,  
    172348,171709,171068,172347,172987,173627,174905,174907,175547,  
    176827,176186,177466,178105,178745,180026,179387,180027,181306,  
    181307,181947,181946,182586,182588,183229,183870,183871,183233,  
    183871,184513,183874,183234,182595,182597,182598,181957,180677,  
    180037,179399,178759,178120,176842,175563,175561,174280,173000,  
    172362,172363,171084,170444,169805,169804,169806,169807,169165,  
    168524,167885,167888,167889,166609,165329,165331,165332,165334,  
    164695,164697,163418,162139,161498,160857,160216,159576,160218,  
    160860,160861,160862,160224,159584,158946,157667,157029,156390,  
    155111,154473,153833,152553,151913,151274,149996,149357,149358,  
    148720,148721,148723,148725,147447,146168,145529,144249,143610,  
    142970,141050,140410,139132,137854,137215,136576,135297,134658,  
    133377,132736,132734,132092,131451,130809,130168,130806,130164,  
    130162,130160,129520,129519,128877,128236,128235,129513,128233,  
    128231,128870,128868,127589,126951,126311,126310,123747,123106,  
    123104,123103,122461,123099,122458,122457,121815,121175,120534,  
    120532,120531,119250,119249,118608,118609,118608,118607,118606,  
    119244,119243,118602,118600,118599,118598,117958,117956,117315,  
    117953,118593,119872,119232,118593,117952,117313,117312,117310,  
    117949,117948,117947,118585,119225,119864,121143,120502,119860,  
    119219,119858,119856,120495,119854,119213,118571,117931,117291,  
    115371,114732,114091,114089,114088,114086,114084,114083,113442,  
    112803,111523,110884,109604,108965,108323,108322,108960,109599,
```

110239, 111516, 110875, 111514, 111513, 111512, 111511, 110869, 110228,
109587, 108947, 107026, 106386, 105106, 104467, 103187, 102547, 101908,
101270, 100631, 100633, 99994, 100636, 100637, 101278, 101279, 100641,
100000, 99361, 100003, 100004, 99366, 100008, 100649, 100010, 101292,
102573, 103213, 103854, 104495, 105136, 104497, 101296, 100015, 99375,
98096, 97457, 96819, 96180, 95542, 95543, 94264, 94263, 93625,
92985, 92344, 91703, 90423, 89785, 90424, 91705, 91706, 90426,
90426, 89788, 87870, 87872, 87233, 87234, 87235, 86594, 85954,
85315, 84677, 84678, 84039, 83400, 83402, 83403, 82764, 82765,
83404, 84042, 85322, 84684, 84046, 84048, 83409, 83411, 82772,
82133, 82132, 80852, 81491, 82770, 82768, 82767, 82126, 82125,
81484, 80844, 80203, 79562, 79564, 78925, 78285, 78284, 78282,
78281, 78280, 78919, 78918, 79556, 80835, 80833, 80194, 79556,
78278, 77639, 77640, 77001, 76362, 76363, 76365, 76366, 76367,
76368, 76369, 77011, 77013, 75735, 75097, 75099, 74460, 73180,
71899, 71898, 70616, 70617, 69975, 69332, 68692, 68051, 66770,
65490, 64849, 63568, 62927, 61646, 61005, 61004, 62284, 62923,
62922, 64199, 62918, 62917, 62276, 60996, 59716, 59076, 59075,
59073, 58432, 57151, 56510, 55869, 56507, 56505, 55864, 55862,
56501, 57141, 57782, 58421, 59701, 60982, 61622, 62261, 62260,
63541, 64182, 64822, 66103, 66743, 68023, 68662, 69301, 69940,
69938, 71219, 72499, 73139, 73780, 74419, 75058, 75057, 73775,
72493, 71854, 70573, 69294, 69292, 69290, 69289, 68647, 68005,
68004, 67363, 66722, 66080, 66078, 66077, 66076, 64795, 64155,
63514, 62873, 63512, 62231, 61591, 60311, 59672, 59032, 57753,
57114, 56475, 55195, 55197, 54559, 53919, 52640, 52642, 51364,
50085, 48805, 46887, 46245, 45608, 46891, 45613, 44975, 43055,
42413, 41133, 39855, 37935, 37933, 36652, 36010, 36008, 37928,
39848, 41128, 42406, 43686, 43685, 42403, 41123, 39843, 37922,
39840, 38559, 37277, 36635, 34717, 33436, 32155, 30873, 28952,
27673, 26393, 25754, 25756, 24477, 22559, 20638, 19994, 21270,
22550, 24470, 26391, 27671, 29591, 30229, 32149, 34068, 35990,
37273, 38551, 40472, 41750, 43029, 42388, 40467, 39830, 37908,
35986, 37904, 38543, 39826, 41104, 43024, 43022, 42379, 41737,

40454, 39172, 39810, 39807, 41086, 41728, 43007, 42365, 41724,
42361, 42999, 42357, 42355, 41077, 39796, 39154, 38511, 37868,
37226, 35944, 35941, 37860, 37219, 35299, 37217, 36575, 35295,
33372, 35291, 35930, 34649, 35287, 36564, 37203, 36561, 37199,
38478, 39758, 38476, 37834, 36552, 36549, 35908, 35266, 35264,
35262, 34620, 34618, 33975, 33973, 33970, 32689, 32686, 30762,
32680, 32677, 33955, 34593, 35871, 37150, 38429, 39067, 39065,
40984, 41626, 42908, 44189, 44831, 46112, 46749, 46109, 46107,
46745, 47383, 48661, 48662, 48664, 50584, 51227, 50589, 51231,
50592, 49954, 51233, 52514, 53154, 53791, 54430, 53788, 54427,
56346, 57625, 58266, 59547, 60828, 61469, 61470, 60832, 61473,
62753, 62754, 62756, 63397, 63399, 62760, 64040, 65320, 65958,
66597, 67236, 67875, 67873, 68511, 69150, 69148, 69787, 69789,
69150, 69152, 68513, 68515, 67876, 67878, 67238, 66600, 65961,
65322, 64684, 64046, 62767, 62126, 61488, 60849, 59569, 58930,
58292, 58293, 59571, 60211, 61490, 62130, 62131, 61493, 60854,
60856, 59576, 58936, 58938, 58939, 59580, 60222, 60864, 60866,
60867, 60869, 61510, 60872, 62152, 62153, 62795, 63436, 64077,
64718, 65999, 66639, 67280, 66640, 65360, 64720, 64079, 63440,
64080, 65361, 66001, 65362, 66003, 66002, 67281, 68563, 69204,
67924, 67283, 67925, 69206, 69847, 69849, 70488, 71127, 71768,
72409, 73050, 73691, 74332, 74973, 75614, 76255, 76257, 76898,
76899, 77540, 76901, 78181, 78822, 79462, 80102, 79461, 79460,
78818, 79458, 80739, 82019, 82021, 82019, 82660, 83939, 85219,
85858, 87779, 90340, 90981, 91621, 91623, 92263, 92903, 94825,
95467, 95468, 96109, 96751, 97392, 98672, 99954, 100594, 101236,
101877, 102516, 102515, 103157, 103158, 103799, 104440, 105080, 105721,
106363, 107004, 107005, 105724, 105723, 104442, 103801, 102520, 101879,
101238, 99956, 99316, 98676, 98677, 98678, 99959, 101240, 101882,
102523, 103164, 103805, 105087, 106369, 107011, 107652, 110213, 110854,
111496, 111497, 111498, 112139, 112140, 112782, 112784, 113426, 113427,
114068, 114069, 113430, 113431, 113433, 114074, 114716, 115357, 115359,
116000, 116002, 116004, 117286, 117927, 118567, 119208, 119209, 119850,
120491, 120492, 120493, 121135, 121776, 121137, 120498, 119859, 120500,

121141, 121782, 123062, 123702, 124342, 125621, 125620, 126260, 126899,
127537,

-1, 39039, 37757, 37114, 36472, 36470, 36467, 35825, 35822,
37742, 39023, 39662, 40301, 39020, 38378, 37098, 37100, 35818,
37097, 37736, 37093, 37091, 37088, 37085, 36443, 35163, 33883,
32601, 32599, 32596, 32594, 32592, 33230, 31950, 31309, 31307,
30026, 28105, 28103, 28100, 26817, 26815, 26172, 26169, 27450,
27447, 29367, 30648, 30645, 30642, 30001, 30639, 31278, 30636,
29355, 30634, 32554, 33192, 32551, 31269, 29348, 28069, 26150,
24228, 23585, 22941, 22938, 24217, 25498, 25496, 25492, 24850,
22930, 22928, 22285, 22923, 23561, 21000, 19721, 19078, 21000,
22920, 22918, 22277, 21635, 22274, 23555, 24193, 24830, 24829,
22910, 22272, 20354, 19076, 17799, 16521, 14602, 12681, 12039,
11397, 10754, 10751, 12028, 10110, 10108, 10105, 8828, 8184,
8821, 10099, 12019, 12015, 12656, 13291, 12652, 13288, 13285,
14563, 15200, 15837, 17115, 18393, 20313, 21594, 22231, 22868,
22865, 23502, 25423, 27343, 27985, 29266, 29904, 28623, 27981,
27979, 26697, 27976, 28618, 29257, 28616, 29254, 30535, 31818,
31816, 31173, 29253, 27974, 26053, 24772, 26693, 27973, 28612,
29250, 31169, 32451, 33731, 35011, 36930, 38851, 38853, 40132,
41412, 43332, 43971, 45250, 46528, 47167, 45885, 45247, 43968,
42689, 41409, 40130, 38848, 37568, 35648, 33728, 32448, 30527,
29248, 27969, 26689, 25406, 23486, 24124, 25403, 27322, 28601,
30521, 31798, 33078, 34359, 35638, 36918, 37560, 38841, 40122,
41401, 40760, 39478, 38837, 38195, 38193, 36911, 36909, 35626,
34344, 34983, 36264, 36906, 38187, 38828, 40107, 40106, 40744,
38824, 39463, 40741, 40099, 40737, 40735, 38815, 40094, 40732,
41370, 42008, 42647, 43285, 44565, 45203, 45202, 43919, 42640,
42642, 41362, 40721, 40718, 40076, 41358, 43278, 44557, 45198,
46478, 47117, 46476, 46474, 47113, 48391, 49030, 49671, 50952,
50950, 50308, 49666, 49665, 51586, 51587, 52867, 52865, 51583,
50941, 49661, 49021, 47741, 46459, 45818, 44537, 45178, 45820,
46462, 46464, 47106, 47107, 47110, 46472, 45833, 43913, 43272,

41990, 41349, 40067, 39425, 38783, 38141, 37498, 36858, 36216,
36855, 36213, 34930, 35573, 34295, 33653, 32371, 32369, 34287,
32367, 33645, 31725, 32364, 33643, 32361, 33639, 34918, 35555,
35553, 36832, 37470, 38750, 38751, 40029, 38749, 40028, 38108,
39387, 40025, 40666, 41304, 42582, 41304, 41306, 41308, 42586,
43865, 45144, 45783, 47063, 48342, 49621, 50259, 51538, 52817,
53456, 52815, 54095, 54732, 56011, 56649, 57288, 58568, 59848,
60488, 61129, 62409, 62410, 63689, 64332, 64973, 64335, 63057,
62418, 63699, 64980, 65621, 66261, 67542, 68823, 68824, 68825,
67545, 67547, 67548, 66908, 65629, 64989, 64349, 63071, 62432,
61792, 60512, 59871, 59230, 58590, 57310, 56670, 54752, 54113,
53475, 52836, 51558, 50917, 48997, 48998, 48361, 48362, 48363,
49004, 50284, 50283, 51562, 52841, 54119, 54758, 56037, 56677,
57958, 59237, 59879, 61161, 61163, 60524, 59886, 59888, 59890,
60531, 60532, 61173, 61171, 61810, 62447, 61805, 61804, 62442,
62441, 63081, 64362, 64363, 65003, 66283, 66281, 64999, 65638,
66277, 67557, 68837, 69476, 70115, 70754, 70753, 70112, 70111,
70109, 70748, 70746, 71385, 70743, 70103, 70741, 71379, 71378,
70737, 69456, 68817, 68177, 67539, 67538, 66258, 65617, 66255,
67534, 68174, 68814, 70095, 70735, 71375, 72014, 72012, 72010,
72649, 73290, 73929, 72648, 73928, 74566, 75205, 75844, 75843,
77122, 77121, 78400, 77758, 77757, 77756, 78397, 79037, 79036,
79035, 79033, 79031, 79671, 80313, 80314, 80315, 80956, 81596,
82878, 83518, 84157, 85437, 85436, 85435, 85433, 85432, 85430,
85429, 85428, 85427, 85425, 86064, 86704, 87344, 89264, 90543,
91184, 92464, 93106, 92467, 93108, 93750, 93111, 93112, 93114,
93115, 93116, 91837, 91838, 91200, 89919, 89280, 88641, 88003,
87364, 86725, 86086, 85448, 86089, 86090, 86091, 85453, 84814,
84175, 84816, 84817, 86739, 86740, 88021, 88024, 88665, 89306,
89307, 90588, 91228, 91867, 91229, 89949, 89310, 89951, 88671,
88670, 88028, 87387, 87385, 86744, 85464, 84822, 84181, 83541,
82903, 83543, 83544, 83546, 84186, 85467, 85469, 86110, 86751,
86752, 87394, 88674, 89955, 91236, 91878, 91240, 91881, 91883,
91241, 89319, 88680, 89321, 88683, 88684, 88685, 88687, 89326,

88687, 88689, 88691, 88692, 89330, 89329, 89328, 89327, 89966,
 89967, 91247, 91887, 92528, 93169, 93811, 93814, 93175, 93816,
 93818, 93819, 93181, 93182, 93183, 93823, 94463, 95103, 96382,
 97661, 98940, 98938, 99577, 98936, 98935, 98934, 98933, 99571,
 98930, 98929, 98927, 98926, 98925, 98924, 98283, 98282, 98281,
 97639, 97638, 97637, 97636, 98275, 98915, 99555, 100193, 99552,
 99551, 98909, 98908, 98267, 97626, 97625, 97624, 97622, 97621,
 96979, 96338, 95698, 94419, 93778, 93139, 93138, 92496, 93135,
 93133, 93131, 93129, 93127, 93126, 93125, 93123, 93761, 94400,
 94398, 95036, 94394, 95032, 94390, 94389, 95028, 96307, 96306,
 96945, 97584, 98223, 98862, 99502, 100782, 101421, 101420, 102058,
 102696, 103336, 103974, 104614, 105893, 106532, 107171, 107810, 109089,
 109731, 110370, 111651, 112291, 113570, 114849, 115490, 116770, 117411,
 118693, 119335, 119976, 120616, 121257, 121899, 122541, 123183, 123825,
 124466, 123828, 123830, 123191, 123193, 123834, 123836, 123198, 123199,
 123201, 122563, 122564, 122565, 122566, 122568, 124490, 124492, 123854,
 123855, 125137, 127056,

-1, 128336, 128975, 129616, 130257, 130898, 131539, 132181, 132821,
 134102, 135383, 136023, 136663, 137304, 139223, 139862, 141781, 142420,
 143700, 144341, 144982, 146263, 147544, 148185, 149465, 150105, 151386,
 152667, 153307, 154589, 155230, 156510, 157151, 159072, 160352, 160353,
 160995, 160357, 160358, 159720, 160361, 160363, 159724, 159725, 159727,
 159088, 159089, 158451, 157172, 156533, 155894, 154615, 154617, 153338,
 152058, 151418, 150779, 150781, 150142, 149502, 148222, 147582, 145661,
 145662, 145024, 143745, 143747, 143109, 142470, 141831, 141192, 139271,
 137991, 137351, 136710, 134149, 132869, 132229, 130950, 129672, 129034,
 128395, 127756, 127117, 126479, 125841, 125202, 124563, 123925, 123286,
 122647, 122007, 120728, 120089, 119449, 118170, 117530, 117529, 118167,
 118166, 118165, 118164, 118803, 118801, 118799, 118797, 118156, 116876,
 116235, 115593, 114952, 114951, 114310, 113669, 111748, 111106, 109825,
 109186, 107905, 107904, 107263, 106623, 105342, 104701, 103420, 102779,
 102138, 101497, 100217, 101498, 102139, 102780, 102141, 102782, 103423,
 104705, 105346, 105987, 107268, 107909, 108549, 109831, 111112, 111753,

112394, 113675, 114315, 114956, 116236, 116877, 116879, 116241, 116242,
 116244, 115605, 114967, 114969, 114330, 114332, 113054, 113056, 113058,
 112419, 111780, 111141, 111142, 109862, 109863, 109224, 108585, 107945,
 106664, 106663, 106661, 105380, 104100, 104738, 105377, 106016, 106014,
 106652, 106651, 106010, 105371, 104091, 105369, 104728, 103448, 102807,
 102166, 100885, 100246, 100247, 100248, 100889, 102170, 102812, 103454,
 104095, 104097, 103458, 103459, 103461, 104741, 104742, 104743, 104744,
 104746, 105387, 105389, 105390, 105392, 105394, 105395, 104757, 105398,
 106039, 106680, 107321, 107323, 107324, 107963, 107962, 108603, 109245,
 109246, 109247, 109889, 111169, 113730, 114370, 115012, 116292, 116933,
 117574, 118854, 119495, 120776, 120778, 120140, 118860, 117581, 116942,
 114382, 114383, 113745, 112467, 111828, 111829, 110550, 110552, 109913,
 109274, 108635, 107996, 108638, 108640, 108001, 108002, 108643, 109284,
 110565, 110566, 112487, 114408, 114409, 113770, 113132, 113773, 114413,
 116334, 118255, 119534, 120814, 121455, 122096, 122737, 124018, 124658,
 125299, 125940, 126582, 127223, 127225, 125303, 124023, 123382, 122741,
 122100, 122099, 121458, 120817, 119535, 117616, 116977, 116339, 116980,
 117622, 118262, 118903, 118905, 119546, 120186, 120187, 120188, 118909,
 118910, 118271, 118273, 117633, 115713, 115073, 113791, 113150, 112508,
 110587, 109309, 109310, 108672, 108674, 109956, 109315, 108677, 108679,
 108040, 108041, 108043, 107404, 107406, 106768, 106129, 105490, 104851,
 104212, 103573, 102934, 102295, 101655, 100375, 100374, 99735, 99095,
 99096, 97815, 96533, 95892, 95251, 94612, 93974, 93335, 93337,
 92696, 92695, 92693, 93332, 92691, 92049, 91408, 90769, 90771,
 89493, 88855, 88856, 89496, 90135, 91415, 90776, 90138, 90140,
 90141, 90142, 92061, 92062, 92063, 93344, 93343, 93984, 95264,
 95905, 95266, 95268, 95269, 93989, 93349, 92068, 91427, 90785,
 90145, 89507, 88869, 88230, 87590, 86951, 86313, 86314, 86956,
 86317, 86318, 85679, 84401, 83763, 83124, 81845, 81205, 79927,
 79288, 78649, 77369, 76729, 76089, 74810, 74170, 72890, 72249,
 71608, 70967, 70966, 72245, 72243, 70964, 70963, 70961, 70320,
 69681, 69042, 68403, 67764, 67125, 65847, 65208, 63930, 63931,
 62653, 62654, 62656, 62658, 62660, 62661, 62662, 62664, 62024,
 62026, 62027, 62669, 62668, 63309, 63311, 62673, 62675, 62033,

192022* 192044* 192029* 192021* 192020* 192044* 194433* 194432* 194432*
61394, 60115, 60116, 58837, 58198, 57560, 57561, 57563, 57564,
58843, 60124, 59485, 58846, 58208, 57569, 56929, 56289, 55651,
56931, 57570, 58850, 60128, 60127, 62045, 62684, 63323, 63962,
65240, 65239, 65238, 66517, 67156, 67796, 68435, 69715, 70996,
72276, 73556, 74197, 74837, 76117, 75479, 74840, 74201, 73561,
72922, 72923, 71643, 70366, 70367, 69086, 68447, 68449, 67169,
66528, 65249, 65248, 64607, 63968, 63329, 62689, 61409, 61411,
61412, 60773, 60774, 61414, 60776, 60137, 60138, 60140, 60781,
60142, 60144, 59505, 58867, 57588, 57589, 56951, 56952, 56314,
56316, 56957, 55678, 53757, 53116, 51835, 51195, 51197, 50558,

-1, 250291, 248369, 244524, 241321, 238759, 238121, 238765, 236845,
236201, 234919, 232998, 231078, 230441, 231086, 233648, 234932, 233657,
231740, 229180, 227900, 225979, 225984, 224707, 225350, 224071, 224074,
222797, 222159, 222163, 222805, 224087, 225369, 226652, 226654, 225374,
223455, 222177, 220898, 219621, 218340, 218342, 219624, 220266, 218986,
219629, 219632, 220911, 222191, 223473, 224118, 222840, 221560, 220279,
219641, 220281, 221562, 222844, 223487, 222207, 221565, 220285, 220287,
220928, 222207, 223489, 223491, 222854, 222214, 222217, 222220, 222862,
220940, 220301, 219659, 219022, 219024, 217742, 218380, 217097, 217100,
217739, 217743, 218385, 219026, 219666, 219668, 219029, 217749, 217746,
216465, 215823, 215180, 215182, 215185, 215187, 215830, 216470, 217113,
217116, 217759, 217762, 217765, 217768, 217770, 217132, 217775, 217778,
218420, 217140, 216501, 217144, 217146, 217788, 219067, 219070, 218433,
218435, 217798, 217160, 216520, 215240, 214600, 213320, 212039, 211398,
210118, 210120, 209481, 209480, 208201, 207560, 207558, 207556, 206276,
205638, 205639, 206279, 206280, 206282, 205642, 205003, 204364, 203086,
202446, 203088, 201808, 201810, 201170, 200534, 199895, 199897, 199258,
200536, 200538, 201818, 201176, 201815, 201174, 201812, 202451, 203089,
204369, 205008, 204367, 205646, 205645, 206284, 207563, 208204, 209484,
209486, 209488, 210769, 212049, 212690, 213971, 214611, 215250, 216531,
217171, 217813, 218453, 219090, 219731, 221012, 222290, 223570, 223568,
225488, 226766, 228683, 230600, 231244, 231248, 231253, 231897, 233820,
235104, 237028, 238952, 240875, 241521, 241527, 240891, 237695, 235137,

233220, 231303, 230026, 228749, 226832, 225553, 224277, 222357, 221720,
 220443, 219805, 219168, 217891, 217893, 217255, 217258, 215979, 215340,
 214698, 214058, 214059, 214700, 214061, 213422, 214064, 215344, 214706,
 213426, 212787, 212789, 214069, 214071, 214074, 212794, 212155, 212157,
 212798, 213437, 214078, 214720, 214082, 213444, 212806, 212168, 212171,
 211532, 211535, 212177, 211539, 211542, 212185, 212187, 212190, 212192,
 212195, 212198, 212840, 212203, 212204, 212207, 211570, 211573, 210935,
 210936, 210939, 209659, 209020, 209662, 210303, 210945, 210947, 211588,
 210949, 210310, 209033, 208394, 207756, 207758, 207761, 207123, 207125,
 207127, 207768, 207129, 206488, 205209, 204571, 204572, 203935, 204577,
 204579, 205221, 205859, 205861, 206502, 206504, 206507, 207149, 207151,
 207152, 207155, 207797, 207799, 207801, 207163, 208443, 209084, 209725,
 210363, 211003, 211643, 212925, 212287, 211645, 211008, 211010, 211012,
 211653, 211015, 210376, 209738, 209098, 208460, 207822, 207824, 206546,
 206548, 206550, 205913, 205915, 205917, 205919, 205921, 205283, 205284,
 205286, 205288, 205930, 205292, 205933, 205294, 205296, 205297, 205298,
 204660, 204022, 204664, 204666, 204668, 205309, 205311, 205953, 205955,
 204676, 204678, 204039, 204041, 204683, 205324, 205326, 205968, 205971,
 205973, 205975, 205337, 205978, 205980, 205343, 204704, 205346, 205987,
 206628, 205990, 205350, 204712, 204714, 204716, 204718, 204720, 205361,
 205363, 205365, 205367, 206009, 206011, 206014, 206656, 207296, 207297,
 207299, 207940, 208580, 209223, 208584, 209226, 208587, 209230, 209233,
 209875, 210516, 209879, 210521, 211163, 211165, 211805, 213086, 212448,
 211809, 213089, 213092, 213094, 213736, 214379, 215021, 214382, 215663,
 215662, 216942, 217581, 218859, 218856, 220136, 220775, 221413, 222053,
 223331, 223969, 225888, 227168, 229089, 231009, 232930, 234850, 236132,
 238694, 239331, 241246, 244443, 248923, 250843,

-1, 160524, 161165, 161166, 161168, 161169, 160531, 159892, 159894,
 159895, 159896, 159897, 159899, 159260, 158621, 158623, 157985, 157986,
 157988, 157350, 157351, 157353, 157994, 157995, 157996, 158638, 159279,
 160560, 161200, 160561, 159923, 158644, 159924, 160564, 161203, 161204,
 160565, 161205, 161845, 161846, 162487, 163127, 164408, 164410, 165051,
 165053, 165694, 165056, 165057, 165058, 165699, 164422, 164423, 164425,

163786, 162506, 161227, 160588, 159308, 158670, 158031, 156751, 156112,
 154832, 154192, 152271, 151631, 150350, 149708, 149067, 148426, 148425,
 147784, 145861, 145219, 143939, 143298, 141377, 140736, 140735, 139454,
 138813, 138172, 139451, 140091, 141371, 142011, 143290, 143930, 143928,
 143927, 143286, 143285, 142643, 142642, 142001, 141360, 140721, 140082,
 139443, 139441, 139440, 138798, 138797, 138796, 138154, 139434, 139433,
 139432, 139431, 140071, 140710, 141350, 141349, 141987, 140706, 140705,
 140703, 141342, 141981, 142620, 142619, 143259, 143258, 143896, 144537,
 145175, 145814, 145813, 145812, 145811, 146449, 146448, 147086, 147724,
 147723, 148362, 149002, 149641, 150921, 151562, 151561, 152202, 153482,
 154763, 155404, 156684, 157325, 158605, 160524,
 -1, 6590, 7232, 7235, 7875, 8518, 9155, 9793, 10435,
 11077, 11716, 12998, 13641, 13004, 12367, 12370, 14293, 14296,
 16216, 17497, 18779, 21339, 22621, 24541, 26460, 27741, 29660,
 30944, 29665, 31586, 32228, 34148, 32866, 32863, 34783, 36702,
 37984, 37347, 36068, 37989, 39909, 39906, 41185, 43104, 45024,
 46304, 47584, 48866, 50147, 51427, 52708, 53348, 55270, 57192,
 57833, 59114, 59757, 59758, 60399, 61041, 61043, 59763, 59124,
 57844, 56565, 55284, 54646, 53367, 52728, 51448, 49527, 49528,
 48889, 48251, 48254, 47616, 46338, 45699, 44420, 43141, 41862,
 41224, 41226, 40588, 39951, 39313, 38675, 37396, 36758, 35479,
 34837, 34195, 34833, 33555, 31634, 30996, 32276, 33557, 34198,
 34201, 32281, 31001, 29079, 27797, 27156, 27798, 28440, 27161,
 25240, 24599, 23960, 23321, 23323, 22043, 21401, 20125, 18206,
 18204, 16285, 14365, 12444, 12441, 11161, 9883, 10527, 9887,
 7966, 7325, 6045, 5407, 5409,

-1, 29600, 30882, 32803, 34084, 35367, 35370, 36013, 35375,
 36658, 36659, 34741, 36022, 37945, 39864, 38586, 40508, 41150,
 43070, 43711, 44990, 46269, 46907, 48189, 48827, 48825, 48822,
 49460, 50740, 52021, 52024, 51385, 51387, 52029, 52670, 52672,
 53952, 54593, 55235, 55236, 56519, 56520, 57162, 55882, 55241,
 54599, 53958, 52677, 53319, 53961, 54603, 54605, 53325, 52684,

51404, 50123, 49481, 48200, 46280, 47563, 48844, 50126, 48847,
 48209, 46929, 45650, 45009, 43727, 43085, 41804, 41801, 40519,
 38599, 38601, 36679, 35400, 34118, 32836, 30913, 29631, 29628,
 27708, 26427, 27065, 25783, 25781, 27059, 27057, 27695, 26416,
 24495, 22575, 21932, 23210, 25128, 27047, 28327, 29608, 28326,
 27046, 25766, 23847, 22568, 21930, 21928, 21925, 22563, 24481,
 25760, 27680, 29600,
 -1, 129192, 129833, 130474, 130476, 129837, 130476, 130474, 131755,
 131117, 131758, 132400, 132401, 132402, 132403, 133045, 135604, 135605,
 135606, 135607, 135608, 135609, 136251, 136253, 136254, 134975, 134976,
 134978, 135619, 136260, 136901, 137543, 137544, 137546, 137547, 136905,
 136904, 135623, 134341, 134342, 133701, 133700, 133059, 131777, 131775,
 131133, 131132, 130491, 130489, 130487, 129845, 129843, 130482, 131121,
 131119, 130478, 129837, 128557, 128556, 128554, 129192,
 -1, 77109, 76471, 77113, 76474, 76476, 75837, 76478, 76480,
 75841, 75201, 74562, 73282, 72641, 73280, 72000, 71359, 70077,
 68797, 68155, 68154, 66875, 66236, 64956, 64955, 64953, 65592,
 64953, 64314, 63672, 63671, 64310, 65589, 65588, 66229, 66869,
 67510, 68150, 68788, 68789, 68791, 70070, 70072, 70074, 70714,
 71994, 72633, 71991, 73271, 73272, 73912, 74550, 74552, 75194,
 74555, 75193, 75192, 76471, 77109,

-1, 30188, 30830, 32110, 32752, 33395, 33397, 34039, 34677,
 34674, 35312, 37233, 37875, 37878, 39799, 39801, 39804, 38527,
 38529, 37251, 38533, 39175, 38537, 37899, 36618, 36620, 34701,
 34058, 33415, 32134, 29574, 27653, 26372, 24454, 23175, 21893,
 22530, 24449, 26368, 25085, 25083, 24441, 25718, 23798, 23796,
 25074, 25712, 26350, 28268, 30188,
 -1, 84242, 86164, 87446, 88727, 89369, 89367, 91286, 92566,
 92568, 93210, 93211, 93212, 93214, 93215, 91935, 91295, 90655,
 90014, 89373, 88733, 88734, 88736, 88737, 87455, 87454, 88093,
 86813, 86171, 85530, 84248, 84250, 83611, 82973, 83614, 83615,
 83616, 82335, 81695, 81053, 81051, 81048, 81687, 81686, 82325,
 82324, 83603,

-1, 130450, 131732, 133013, 132373, 131093, 130454, 131094, 131735,
 132377, 133017, 133018, 131737, 130456, 129816, 129178, 129179, 129177,
 129175, 129174, 128533, 127894, 127895, 127896, 127898, 127900, 127901,
 127262, 127260, 127258, 127257, 127255, 127894, 127892, 128532, 129172,
 129811, 130450,
 -1, 86768, 87409, 88051, 88692, 88054, 88695, 88056, 87418,
 87419, 87421, 87422, 87423, 88065, 88706, 88708, 88709, 88710,
 88711, 88072, 87433, 86153, 86151, 85510, 84228, 84226, 83585,
 83583, 83582, 84220, 84219, 83578, 83577, 82299, 82298, 82297,
 82296, 81654, 82933, 82932, 84211, 84850, 85489,
 -1, 30491, 31773, 33055, 33698, 33701, 33062, 32419, 30498,
 28578, 27298, 25379, 24101, 22182, 20264, 18986, 17067, 15790,
 14513, 13876, 13239, 12602, 10682, 10040, 10677, 11955, 12592,
 12588, 13865, 15143, 15782, 17058, 18338, 20257, 22175, 23456,
 24734, 26653, 27933, 28571, 30491,

-1, 95912, 96553, 95914, 95915, 95916, 95278, 95920, 96559,
 97200, 95922, 95282, 95925, 95286, 95287, 95289, 94649, 93369,
 92730, 91450, 90811, 90171, 88890, 88890, 88249, 90168, 91447,
 92086, 92725, 93364, 93363, 92723, 93362, 94001, 94638, 94637,
 94636, 95274, 95913,
 -1, 123369, 124009, 124651, 125292, 125933, 126575, 127215, 127857,
 128498, 129778, 130419, 131701, 132343, 132985, 132986, 132347, 131067,
 130427, 130426, 129144, 128504, 127863, 127222, 126580, 126578, 125297,
 124655, 124014, 123373, 123372, 123370, 123369,
 -1, 127233, 128513, 129155, 130435, 131076, 130438, 131078, 131080,
 131082, 131723, 131725, 131086, 129167, 128528, 127249, 127251, 126609,
 125969, 125328, 124689, 124050, 123409, 122127, 122766, 123404, 124043,
 124681, 125320, 125958, 125957, 126595, 126593,
 -1, 48916, 48918, 48919, 48920, 49559, 50837, 50839, 52120,
 52759, 52762, 53404, 54046, 52768, 52770, 52131, 51494, 50215,
 48935, 48293, 46373, 47012, 46371, 47010, 47648, 47007, 47005,
 47003, 48923, 48921, 47641, 46360, 46359, 46997, 48916,
 -1, 182597, 182596, 183235, 183874, 183876, 184515, 185155, 184514,

185153, 184512, 183869, 183228, 183227, 183868, 184509, 185150, 185792,
186433, 186434, 187075, 187078, 187719, 187080, 186441, 186443, 185802,
185160, 184519, 183238,
-1, 147596, 148876, 149517, 150799, 151440, 150802, 150163, 149524,
148884, 148245, 146965, 146326, 145686, 144407, 143767, 141848, 141849,
140568, 138647, 139286, 141843, 141841, 142479, 143118, 144397, 145678,
146318, 146957,
-1, 84003, 83365, 84644, 85284, 85924, 86564, 87206, 85926,
84646, 83367, 83368, 82730, 83372, 84012, 84652, 85294, 84654,
83375, 84016, 84017, 82737, 82096, 82095, 82093, 82092, 82091,
82089, 82087, 82725, 82724, 84003,

-1, 72251, 73531, 74171, 74811, 76092, 76731, 77372, 78012,
79291, 79932, 81211, 82491, 81852, 81853, 81213, 80572, 79933,
78653, 78015, 78656, 77375, 76734, 76094, 75454, 74813, 73534,
72894, 72253, 71613, 70972, 72252,
-1, 79574, 80216, 80217, 80219, 80220, 80221, 80860, 80861,
80864, 81505, 80865, 80226, 79585, 80224, 78945, 78944, 78304,
77662, 77661, 77660, 77019, 76379, 75740, 74461, 75099, 76378,
77017, 77657, 78296, 78936, 79574,
-1, 16110, 16753, 17395, 16117, 17398, 18036, 19319, 18681,
18043, 16765, 17409, 16771, 15491, 13572, 12930, 12287, 10366,
11004, 12283, 12924, 14845, 14842, 13561, 12279, 11637, 11634,
12913, 14191, 16110,
-1, 49920, 48001, 47360, 46082, 47362, 48643, 48644, 48645,
48647, 49927, 50569, 51211, 51852, 50573, 49934, 48654, 48656,
46738, 46096, 44815, 44172, 44170, 42889, 42247, 40965, 40323,
39682, 38400,
-1, 213304, 214585, 215226, 215867, 215869, 217150, 217153, 217155,
216517, 215238, 214598, 213318, 212677, 211396, 210116, 209475, 209473,
210753, 211393, 212674, 213312, 213310, 213307, 213304,
-1, 166388, 167030, 167031, 168310, 168951, 168312, 167673, 166394,
165755, 164476, 164475, 164473, 164472, 163192, 163191, 161909, 161268,
161267, 161907, 163189, 163190, 164470, 165110, 166388,

-1, 174055, 174696, 174697, 175339, 174701, 174062, 173423, 172144,
 172145, 171507, 170866, 170228, 168949, 168307, 167666, 168305, 168944,
 170223, 170222, 171500, 171499, 172138, 173416, 174055,
 -1, 9749, 11029, 12311, 12314, 12956, 14236, 16156, 18076,
 18720, 19363, 19366, 19369, 18091, 18734, 18097, 16818, 15538,
 14255, 14252, 14249, 14885, 14242, 13599, 11678, 11037, 11034,
 9752, 9749,

-1, 108329, 108330, 107691, 107693, 107695, 107697, 108338, 108340,
 108982, 110264, 110265, 109627, 109625, 108984, 108342, 107700, 107699,
 107057, 107056, 107054, 107052, 107690,
 -1, 5287, 5288, 5289, 5927, 6564, 7207, 7203, 8483,
 9125, 9123, 10401, 11681, 12324, 12328, 11690, 12333, 11695,
 12336, 11699, 10421, 9139, 7862, 7221, 7225, 5946, 5306,
 5308, 5313,
 -1, 134267, 134269, 134910, 134912, 134914, 135556, 135557, 135559,
 135560, 135562, 134923, 134921, 134920, 134277, 134276, 134274, 134272,
 133631, 133630, 133628, 134267,
 -1, 46817, 46820, 46823, 47465, 48103, 48742, 50021, 50023,
 48105, 50025, 48748, 48106, 48109, 46830, 46188, 46186, 45547,
 44908, 44266, 45544, 45542, 46180, 46817,
 -1, 86968, 87608, 88248, 87610, 86968, 86970, 86971, 86972,
 87614, 86974, 86336, 86337, 85697, 85056, 85054, 84413, 83772,
 83771, 84411, 85051, 85690, 86329, 86968,
 -1, 5459, 6100, 6743, 6746, 7384, 8664, 9946, 11870,
 9950, 8672, 7393, 6113, 6115, 6756, 7398, 8676, 9320,
 8044, 6761, 6119, 5478, 5473, 5477, 5482, 5485, 5487,
 -1, 113812, 115093, 115094, 116375, 116376, 115737, 117019, 117020,
 116379, 115738, 115736, 115095, 113816, 113177, 111896, 111895, 111894,
 112533, 113173,
 -1, 58352, 58994, 58995, 58996, 58358, 57719, 57720, 56441,
 55803, 55804, 55166, 55164, 55163, 55161, 56440, 56439, 56437,
 55795, 57076, 57075, 58354, 58352,
 -1, 73901, 75182, 74544, 74545, 73907, 73908, 72629, 71989,

70709, 70069, 69427, 69426, 69425, 70705, 70703, 70702, 71343,
72623, 73263, 73902, 73901,
-1, 74743, 75385, 75386, 74747, 74108, 73469, 73471, 72193,
71554, 70914, 69634, 68995, 68993, 70273, 71551, 72190, 72829,
73468, 74746, 74745, 74743,

-1, 53285, 53287, 53927, 53929, 53290, 52651, 52652, 52654,
53935, 52657, 52015, 51374, 50093, 49451, 48809, 46889, 48167,
49446, 50726, 52006, 53285,
-1, 136205, 136207, 136209, 135571, 136212, 136214, 136216, 135578,
135579, 135578, 135576, 135574, 135572, 135571, 135570, 135568, 135567,
135565,
-1, 25097, 26378, 27658, 27660, 28942, 30864, 29586, 28948,
27029, 25108, 24465, 22547, 20626, 21264, 21262, 21900, 23179,
24461, 24458, 25097,
-1, 81436, 80798, 81440, 80802, 80163, 81443, 80804, 81446,
81447, 81449, 80809, 79529, 79527, 78886, 78884, 78243, 78881,
80160, 80159, 80797,
-1, 28257, 29540, 32101, 30823, 30825, 28907, 26348, 25070,
23793, 23155, 21872, 19951, 19947, 18664, 19942, 20579, 23140,
25059, 26978, 28257,
-1, 111547, 111548, 111550, 111551, 112193, 111555, 111556, 111558,
110917, 110916, 110274, 110272, 110271, 110270, 111551, 111549, 111548,
-1, 122136, 121497, 121499, 121500, 122780, 122781, 123422, 122782,
122783, 122144, 120864, 119582, 120221, 120860, 120859, 120857, 121496,
-1, 11013, 12294, 13575, 15497, 14859, 16781, 16785, 14866,
12946, 11666, 11023, 11020, 11658, 12938, 11657, 11015, 11013,
-1, 12259, 13541, 13543, 13545, 12267, 11628, 12271, 10993,
9712, 8434, 7792, 9071, 8428, 9066, 10344, 11622, 12259,
-1, 5251, 5255, 5893, 6535, 6538, 6540, 7822, 7184,
7187, 7189, 5266, 6544, 6543, 5902, 5259, 5257, 5254,
5252, 5251,
-1, 133063, 133704, 133705, 133707, 133069, 131790, 131788, 132429,
133067, 133066, 133065, 133064, 133063,

-1, 167617, 168897, 170177, 170818, 171460, 170181, 169542, 168903,
168261, 168260, 168259, 167618, 167617,

-1, 71823, 73104, 74385, 74387, 75027, 75667, 76308, 75029,
73747, 73107, 72466, 71826, 71185, 71184, 71823,
-1, 82301, 82942, 82943, 82945, 83586, 82946, 81667, 81028,
81029, 81028, 81026, 81665, 81663, 81662, 82301,
-1, 16563, 18485, 18487, 17849, 17211, 17853, 17215, 15937,
14015, 14014, 12731, 14650, 12726, 14004, 16563,
-1, 72460, 73100, 73102, 74381, 74383, 75024, 76304, 76306,
75664, 74384, 74383, 73742, 73103, 72461, 72460,
-1, 83010, 83651, 83011, 82372, 81734, 81736, 82378, 81739,
81738, 81736, 81734, 81733, 81731, 83010,
-1, 83623, 85543, 85544, 85545, 85547, 84908, 83629, 82988,
81708, 81707, 81706, 82345, 82984, 83623,
-1, 76252, 76893, 76894, 77535, 78177, 78819, 79460, 78180,
77539, 76898, 76257, 76255, 76254, 75612,
-1, 87211, 87852, 87853, 87855, 87216, 87217, 86578, 86579,
85937, 86576, 86575, 87213, 87212,
-1, 181971, 182612, 181974, 182615, 181977, 181336, 181335, 181334,
181332, 181972, 181971,
-1, 129656, 130298, 130299, 129659, 129020, 127740, 127738, 127737,
128376, 129656,
-1, 137499, 136860, 136222, 136224, 135585, 135584, 135582, 135581,
136220, 136859,
-1, 120781, 122701, 122703, 122704, 122065, 120784, 119503, 119501,
120141, 120781,
-1, 118936, 118937, 120218, 119580, 118940, 118939, 118298, 118297,
117656, 118296,
-1, 57341, 58622, 59262, 58624, 57343, 56063, 55422, 55421,
56062, 56701, 57341,
-1, 91861, 92503, 92504, 93145, 93146, 92506, 91867, 91866,
91865, 91864, 91862,

-1, 97189, 97831, 98471, 99111, 99112, 99113, 97833, 97193,
 96552, 96551, 97189,
 -1, 5759, 5450, 5140, 5120, 195200, 250240, 250549, 250859,
 250879, 59519, 5759,
 -1, 85938, 85300, 85301, 85942, 85303, 85304, 84663, 84662,
 84659, 85298,
 -1, 58613, 59894, 60534, 61175, 60536, 60537, 59258, 58616,
 57974, 58613,
 -1, 70483, 71763, 73045, 74327, 73046, 72406, 71765, 71126,
 71125, 70483,
 -1, 63482, 63484, 62205, 62207, 62209, 62208, 62207, 61566,
 62204, 63482,
 -1, 21937, 23216, 25137, 25779, 25141, 25143, 22583, 21941,
 21939, 21937,
 -1, 198611, 198612, 197974, 197975, 197337, 197335, 198612, 198611,
 -1, 120848, 120849, 120210, 119571, 118292, 118931, 119570, 120208,
 -1, 138620, 139260, 140541, 141182, 139261, 136701, 137340, 138621,
 -1, 7536, 6899, 6903, 6906, 5626, 5624, 5621, 5619,
 5618, 6897, 7536,
 -1, 65965, 66606, 65967, 65329, 64688, 64049, 64047, 64686,
 65965,
 -1, 77005, 77646, 78288, 78289, 78290, 77648, 77007, 77006,
 77005,
 -1, 67336, 66698, 66058, 65418, 64779, 64138, 64777, 66057,
 67336,
 -1, 15300, 17221, 18504, 18507, 16587, 15944, 15941, 15300,
 -1, 185212, 185213, 185854, 186496, 185215, 185214, 185212,
 -1, 117020, 117661, 118300, 118301, 118302, 117022, 117020,
 -1, 131107, 131108, 131109, 131111, 130469, 130468, 130467,
 -1, 146531, 147813, 147814, 148456, 147815, 147173, 146532,
 -1, 110592, 111872, 111874, 111236, 110596, 109955, 110593,
 -1, 212664, 212666, 212668, 211389, 210747, 211385, 212664,
 -1, 130507, 131148, 131149, 131790, 131791, 131150, 131148,

-1,43062,44343,43705,43066,41787,41144,43062,
 -1,7798,8438,8441,7804,7164,7162,7160,7798,
 -1,111540,112182,111543,112184,111543,111541,
 -1,53135,53776,54418,53780,53778,53137,53135,
 -1,106773,108053,106775,105496,105494,106133,
 -1,60182,60823,60824,60826,60185,60183,60182,
 -1,97194,97835,97197,96558,95917,96556,96555,
 -1,168508,170427,170429,169789,169149,168508,
 -1,115543,116183,116825,116187,116185,115543,
 -1,81485,82127,82128,81489,81488,81487,81486,
 -1,95097,95738,95099,94461,94460,95098,95097,
 -1,68819,69459,69460,69461,68822,68180,68819,
 -1,127265,127906,128547,127908,127267,126627,
 -1,177722,179002,179003,178364,178363,177722,
 -1,6518,6521,6522,6525,5886,5883,5879,6518,
 -1,23609,24251,24894,23614,21692,22330,23609,
 -1,234215,234218,232941,232938,231655,234215,
 -1,5272,5274,6556,6561,5923,5285,5287,5284,
 -1,95081,95083,95085,95086,95084,94443,94441,
 -1,88654,89274,90576,89937,89297,88016,88654,
 -1,16788,17430,18073,16153,14872,14870,16788,
 -1,91877,93158,93159,93160,92520,91879,91877,
 -1,84420,84421,83782,83144,83782,84420,
 -1,88508,88509,88510,88512,88510,88508,
 -1,40086,39448,38169,36887,38166,40086,
 -1,67377,67378,68017,68018,67379,67377,
 -1,56491,56492,55214,55213,55212,56491,
 -1,228533,229176,227898,227256,228533,
 -1,143356,143358,143359,142718,143356,
 -1,139532,140174,140175,140174,139532,
 -1,250275,247075,243234,240673,240668,
 -1,129786,130428,131069,129788,129786,
 -1,111560,111562,111563,111561,111560,
 -1,133491,134132,135414,134774,132211,

-1,201805,202447,201169,201808,201805,
-1,138790,138791,138793,138152,138790,
-1,116373,117014,117015,116375,116373,
-1,55857,57137,57139,55859,55857,
-1,72494,73135,73136,72495,72494,
-1,65587,64948,63668,64307,65587,
-1,7827,8470,8474,7836,7191,7827,
-1,7812,9093,9735,8454,7814,7812,
-1,5324,5323,5318,5955,5951,6590,
-1,14827,14829,12910,13548,14827,
-1,71702,71703,71704,71063,71702,
-1,86735,87375,87376,86736,86735,
-1,5606,5607,5611,6254,5617,5617,
-1,32000,32641,31365,30083,30080,
-1,82732,82733,82094,82733,82732,
-1,240668,243230,245149,250269,
-1,105140,106422,105781,105141,
-1,119250,119251,118611,118610,
-1,199259,199902,199260,199259,
-1,136851,136852,137493,136212,
-1,7154,7796,7797,6517,7154,
-1,89923,90565,89925,89923,
-1,76755,77396,76116,76755,
-1,20408,21689,21690,20408,
-1,63969,63331,63330,63969,
-1,16774,16776,15494,16774,
-1,15646,16926,16928,15646,
-1,65632,66272,64993,65632,
-1,135563,135564,135563,
-1,195437,195440,194798,
-1,131103,131745,131103,
-1,162482,162484,161842,
-1,129825,129827,129186,
-1,175977,175338,175977,

-1, 128551, 128552, 128551,
-1, 108841, 108843, 108841,
-1, 128560, 129201, 128560,
-1, 146977, 146978, 146977,
-1, 134943, 134944, 134943,
-1, 129135, 129775, 129135,
-1, 127212, 127853, 127213,
-1, 116324, 116964, 117604,
-1, 135580, 135582, 135580,
-1, 116894, 117535, 116894,
-1, 134952, 134313, 134952,
-1, 134317, 133678, 133677,
-1, 136283, 136924, 136283,
-1, 137566, 137568, 137566,
-1, 102050, 101411, 102050,
-1, 170886, 170887, 170886,
-1, 144634, 144636, 143995,
-1, 131071, 130432, 130431,
-1, 102054, 101415, 102054,
-1, 108198, 108840, 108198,
0
};

Index

Specific routines are listed under the program in which they appear.

- ** 143
- absolute addressing modes 272
- accstart 41, 164
- ACCSTART.S 7
- accsup.o 164
- address errors 223-24
- address mode field 267
- address of handle 22
- address register direct mode 271
- AES viii
- Alcyon C 5, 45
- annotated disassembler 263
- archiver program 45
- arguments 236
- arrow icons 33
- asm 227
- Atari ST Software Developer's Kit* vii, 5, 19, 45, 49, 127, 219
- batch file 5
- BIOS viii, 183
- blitter chip 185
- bomb information 251-52
- bombs 223-24
- boxes 96
- braces 4
- break 4
- brk function 41
- bugs 52
- building a menu tree 96-101, 117
- building an interactive sound control panel 186-203
- build_tree. *See specific program entry*
- bus errors 223
- C compilers viii
- central processing unit (CPU) 223
- CHK instruction error 224
- clearing work area 39
- clock 212
- column 160
- command interpreter 127
- "Command Shell" 127-71
 - build_tree 167-69
 - built_in 142-44
 - calc_dir 160-62
 - call_sys 136-41
 - configac.c 129
 - configap.c 167-68
 - dir_list 146-49
 - do_copy 150-52
 - do_dir_window 157-60
 - doit 162
 - doit.c 129-30
 - do_main_menu 169-70
 - do_menu 169-70
 - do_move 152-54
 - do_rm 154
 - do_title 155-56
 - findcmd.c 141-42
 - get_head 149
 - give_help 135-36
 - got_key 130-35
 - isprg 136-38
 - justdraw.c 129-30
 - linkacc.arg 164
 - linkacc.bat 165
 - link.arg 170
 - linkit.bat 171
 - new_window 162
 - pad 162
 - print_dir 163-64
 - prntfile.c 155-56
 - redo_dir 163
 - save_last 144-46
 - set_screen 136-38
- compiler modifications 6-8
- config.c. *See specific program entry*
- connecting routines 11
- control statements 3
- copying files 150-52
- C routine, calling 234
- cursor, text 15
- dagger 180
- data files 21
- data register direct mode 271
- data transfer address 146
- "Bombsite!" 224-61
 - bomb_info 251-54
 - bt 248-50
 - configac.c 225-26
 - debug.h 257-58
 - decode_status 254-55
 - doit 239-41
 - do_new_window 254-56
 - errors.c 227-29
 - get_args 236-37
 - get_base 246-47

- getbyte 238-39
- getlong 238-39
- get_name 250-51
- get_real 233-36
- getshort 238-39
- get_syms 244-46
- get_trace 229-31
- give_help 242-43
- got_key 241-42
- ismatch 247-48
- just_draw 239
- linkone.arg 259
- linkone.bat 258
- open_data 226-27
- set_top 227
- showwnd 256-57
- trace 243-44
- debugging 223-59
- deleting files 154
- desk accessory programs 15, 127
- desk menu 92
- dialog box 116, 200
- dir_char 147, 150
- disassembler 239, 258, 263, 264
- disassembler portion of "Bombsite!" 263-99
 - addrmode 267-70
 - disassem 274
 - effadd 271-72
 - get_dis 265-66
 - hash_tab structure 275
 - immediate 273-74
 - linkst.bat 299
 - matchB 266-67
 - pcabsimm 272-73
 - setup_dis 274-75
 - tables.c 275-98
- disassembly listing 274
- do_menu. *See specific program entry*
- doit. *See specific program entry*
- drawing a bar chart 78-81
- drawing a line chart 70-72
- drawing pie charts 81-84
- drawing the screen 38
- drawing the Mandelbrot set 101-5
- dummy exit function 41
- editable text fields 117
- envelope 11
- envelope control bit 209
- envelope generator 209
- envelope library v, vii, viii, 11
- envelope library routines 12-46
 - addit.c 42
 - archive.bat 46
 - bldtree.c 43, 50
 - clip_work 39-40
 - clocktic.c 43
 - close_all 40-41
 - close_window 40-41
 - clr_display 39-40

- config.c 12-14, 50
- do_arrows 33-35
- doclean 44, 50
- do_display 38
- doit 38, 50
- do_menu 31, 50
- do_redraw 32-33
- gotkey.c 44, 50
- hide_mouse 37
- h_touched 36
- just_clear 39
- just_draw 32-33, 50
- main.c 14-16
- mousehit.c 44, 50
- multi 22-26
- newwind 42, 44
- nfparts.h 45
- open_data 21-22, 50
- open_vwork 17-18
- open_window 20-21
- pad.c 44
- pos_slide 36-37
- setup_screen 16-17
- setup_window 19-20
- show_form 26
- show_mouse 37
- slide_pos 35-36
- slidsize.c 45
- vdidata.c program 42
- v_touched 36-37
- was_msg 27-31
- window.h 45
- error-handling routines 227
- everything 143
- exception handling 223
- exception number 251
- exceptions 223, 225
- exclamation points 131
- exit a program 40
- file header 244
- file menu 93
- filename 160, 244
- fractals 87
- frame pointer 228, 229, 232
- function addresses 246
- functions 3
- GEM v, vii, viii
- GEMDOS viii
- gemstart 41, 164, 167, 170
- GEMSTART.S 6
- Giaccess 204, 205, 208
- global variables 4
- got_key. *See specific program entry*
- goto statement 4
- graphics accelerator chip 185
- handle 14, 158
 - address 22
 - virtual workstation 17, 38
 - window 21, 38

- hard disk 57
- hashing 265
- hatching pattern 76
- header file 124, 187
- help menu 94
- illegal instructions 224
- immediate modes 272
- indentation 3
- index 266
- indexing errors 224
- indirect recursion 157
- input 22
- input devices 13
- instruction 224, 236
- instruction lengths 234
- instruments 212
- interface variables 12
- interprocess communication 23
- interrupt priority level 254
- Kernighan and Ritchie standard 5
- keyboard 177
- libraries 5
- library names 5
- Line A graphics interface 87, 102
- linear search 266
- linker 45, 50
- load address 233, 243
- long integers 52
- low memory 183
- malloc () subroutine 7
- Mandelbrot Program's Menu and Submenus figure 89
- Mandelbrot set 87
- "MandelZoom!" 87-124
 - addit 97, 100
 - back_to_first 124
 - build_tree 88, 96-101
 - colors 108-10
 - config. c 88-89
 - coordinates 116-20
 - dialog 88, 115-16
 - copy_first 121
 - do_cleanup 110
 - doit 88, 101-5
 - do_main_menu 91-95
 - do_menu 88, 89-90
 - get_val 120
 - give_help 95-96
 - got_key 88, 113-15
 - just_draw 88, 121-23
 - link.arg 124
 - linkit.bat 124
 - mouse_hit 88, 111-13
 - rest_colors 109-10
 - save_colors 109-10
 - save_screen 121
 - set_val 120
 - time_it 105-7
 - time_print 105-7
- mandlzum. *See* "MandelZoom"
- masking 266
- memory management unit 223

- menu activation 89
- menus 168
- messages 26
- modulars 3
- mouse 15, 37, 38
- mouse_hit. *See specific program entry*
- MOVE instruction 267
- move multiple instruction 227
- "Noise!" 175-219
 - add_slider 192-93
 - all_sliders 193-94
 - bellblock 216-19
 - bld_sliders 189-91
 - build_tree 187-89
 - clock_ticks 215-16
 - config.c 175-76
 - do_black 182
 - do_cleanup 183
 - do_effects 213-15
 - doit 176-77
 - do_main_menu 196-99
 - do_menu 195-96
 - do_rhythm 213-15
 - do_slider 199
 - do_white 180-82
 - drums.h 212-13
 - fill_box 180
 - get_clicks 183
 - got_key 185-86
 - just_draw 177
 - keys.h 179-80
 - link.arg 219
 - linkit.bat 219
 - mode_bit 209-10
 - no_clicks 183
 - noise 210-11
 - noise_enable 205-6
 - open_data 183-84
 - percussion 216-18
 - period 210-11
 - play_note 204
 - print_vals 212
 - put_clicks 183
 - rad_button 205-7
 - rest_state 203
 - save_state 203
 - select_on 204-5
 - set_slider 195
 - shape 210-12
 - show_keys 177-79
 - sliders 199-200
 - sliders.h 187
 - slid_val 207-8
 - sl_set 200-202
 - toggles 204-5
 - tone 208-9
 - tone_enable 205-6
 - volume 209-10
 - which_one 202
- null string 63
- object 96
- object code 45

- open_data. *See specific program entry*
- options menu 93–94
- overflow error 224
- overflow table 265
- packing 51
- pathname 141, 149
- phystop variable 227
- piano keys 179
- pitchs 216
- “Plots and Charts” 61–84
 - bar_chart 78–81
 - config.c 61, 62–63
 - doit.c 61
 - draw_bar 80–81
 - drawbox.c 77
 - extract 67–68
 - getmode.c 77
 - grid 73–78
 - label.c 75
 - line_chart 70–72
 - link.arg 84
 - linkit.bat 84
 - open_data 63–64
 - open_data.c 61
 - pie_chart 81–84
 - range 72–73
 - read_data 65–67
 - scale.c 78
 - select_file 64–65
 - strip_comment 67
 - strval.c 75
- pointers 226
- post processor 5
- Print Values menu selection 212
- printing the directory 162
- privilege violations error 224
- program addresses 247
- program code 231
- program counter 224, 228, 252
- program counter relative modes 272
- programming 5
- question mark 248, 250, 268
- quit a program 40
- radio button 186, 206
- RCS 50, 62
- renaming files 152–54
- resolution 108, 121
- resolution modes 49
- resource construction set. *See* RCS
- return address 229
- re-usability 4
- rhythms, generating with the clock 212–19
- screen 16
- screen memory location 137
- script file 5
- scrolling, screen 33
- setexc 225, 226
- shape register 204
- slider 189, 192, 256
- slider box 33, 35, 35, 195, 207
- slider index 202
- sound chip 175
- sound chip registers 197, 203–12
- sound register values 193–95
- stack 229, 231–33
- stack space 6
- stack trace 248
- Stack Trace Structure, The (figure) 232
- startup routines 6, 41
- status register 254
- subroutine addresses 246
- subwindows 32
- supervisor mode 183, 223, 224, 227
- SUPEXEC 183
- switch statement 3
- symbol table 233, 243, 244, 250
- system reset 225
- TAB character 180
- TEDINFO structure 117
- template string 117
- text editor 45
- text string 117
- time intervals 13
- toggle switches 94
- tone 208
- TOS-Takes-Parameters 21
- transient program area (TPA) 251
- TRAPV instruction error 224
- tree 96
- tree index 202
- .TTP extension 21
- user mode 224
- user stack pointer (USP) 251
- validation string 117
- values 120
- VDI viii, 61
- VDI routines 42
- virtual device interface. *See* VDI
- virtual workstation 14, 17
 - handle 14, 38
 - input parameters 17
- voice register 204
- volume 206, 209
- while 4
- wildcards 146
- window 12, 13, 19, 20, 28, 40, 157, 162, 254
- window handle 21, 38
- window management routines 27
- “World Map” 50–58
 - config.c 50
 - doit 51
 - link.arg 57
 - linkit.bat 57
 - map 52–53
- “World Map” data 303–23
- writing a program 3, 4
- writing mode 76
- zero divide error 224

To order your copy of *COMPUTE!'s ST Applications Guide: Programming in C Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

COMPUTE!'s ST Applications Guide Disk
COMPUTE! Publications
P.O. Box 5038
F.D.R. Station
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 5% sales tax. NY residents add 8.25% sales tax.

Send _____ copies of *COMPUTE!'s ST Applications Guide Disk* at \$16.95 per copy.

Please note: The *COMPUTE!'s ST Applications Guide: Programming in C Disk* is a 3½-inch, double-sided disk.

Subtotal \$_____

Shipping and Handling: \$2.00/disk \$_____

Sales tax (if applicable) \$_____

Total payment enclosed \$_____

☐ Payment enclosed

☐ Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. _____ Exp. Date _____
(Required)

Name _____

Address _____

City _____ State _____ Zip _____

Please allow 4-5 weeks for delivery.

Programming the Atari ST

GEM on the Atari ST is a rich environment, with hundreds of routines enabling you to create sophisticated, powerful applications. GEM's features include pull-down menus, icons, sliders to scroll screen data, mouse-activated screen selections, the ability to move and rearrange windows, and so on.

Written for the C programmer, *COMPUTE!'s ST Applications Guide: Programming in C* contains a set of high-level routines that make working with GEM easier. By using these routines and making them part of your programs, you'll be able to access GEM quickly and effortlessly. Each routine is fully explained in easy-to-understand language.

Also included are a series of six application programs that illustrate how to use the routines to access GEM. These applications show and explain how to use slider boxes, pull-down menus, windows, and much more. The applications included are:

- Plot routines that create pie, bar, and line graphs
- A very fast Mandelbrot set
- A line-drawing routine that creates a world map
- Command Shell desk accessory which allows desktop commands to be used while an application is running
- Sound board simulation using the sound chip
- Debugger utility

COMPUTE!'s ST Applications Guide: Programming in C is for intermediate to advanced C programmers. Written in the clear and concise style that has become the hallmark of all COMPUTE! publications, *COMPUTE!'s ST Applications Guide: Programming in C* includes all the programs to help you access GEM from your C programs. This is a book all Atari ST C programmers will want to add to their reference libraries.

All the programs in this book are ready to type in and use. If you prefer not to type in the programs, however, a companion double-sided disk is available. See the coupon in the back of the book for details.