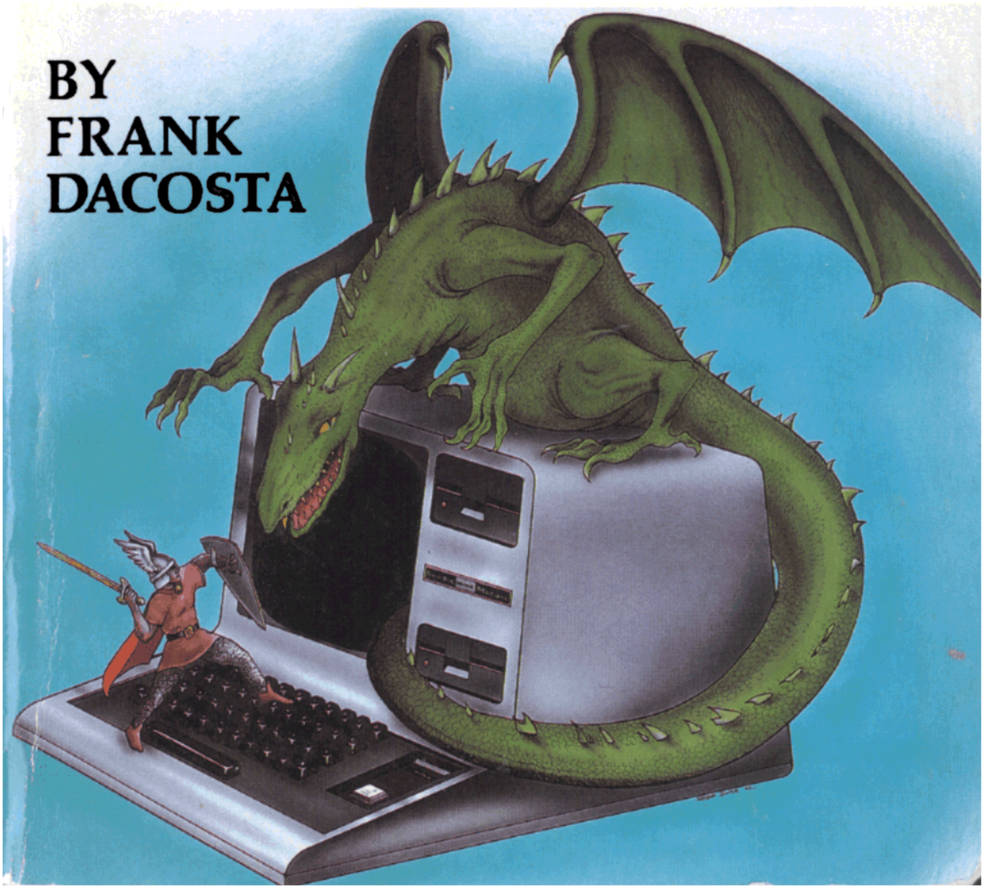
TAB 1422 $9.95

WRITING BASIC ADVENTURE PROGRAMS FOR THE TRS-80

Create exciting computer games while you learn valuable new programming techniques!

BY FRANK DACOSTA



WRITING BASIC ADVENTURE PROGRAMS FOR THE TRS-80

BY FRANK DACOSTA

TAB BOOKS Inc.
BLUE RIDGE SUMMIT, PA. 17214

FIRST EDITION

FIRST PRINTING

Copyright © 1982 by TAB BOOKS Inc.

Printed in the United States of America

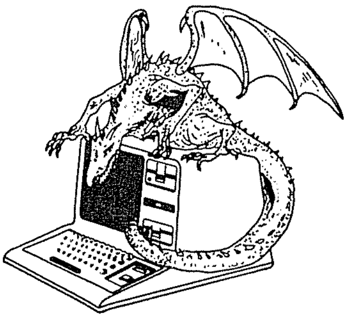
Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

Library of Congress Cataloging in Publication Data

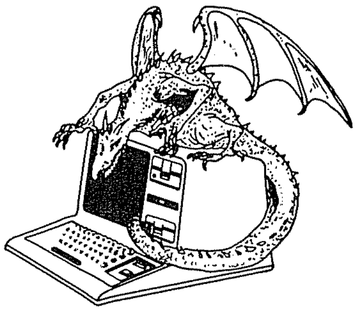
DaCosta, Frank.
Writing BASIC adventure programs for the
TRS-80.

Includes index.
1. Games—Data processing. 2. Fantasy games—
Data processing. 3. TRS-80 (Computer)—Programming.
4. Basic (Computer program language) I. Title.
II. Title: Writing B.A.S.I.C. adventure programs
for the TRS-80.
GV1469.2.D32 1982 794 82-5945
ISBN 0-8306-2422-8 AACR2
ISBN 0-8306-1422-2 (pbk.)

Cover illustration courtesy by Robb Durr.



Contents



Introduction

If you are an adventurer who can find your way out of the most sophisticated mazes, and if you have overcome fantastic threats to win priceless treasures, you are a serious computer adventurer. You have probably thought of writing your own adventure program.

You can begin in BASIC. This book will help you create an expert adventure—with a TRS-80 Model I or III microcomputer, just 16K memory, and a tape system. You need some instruction and some examples, which this book provides. It uses two example programs, Basements and Beasties and Mazies and Crazies. Their organization—from execution loops, to subroutines, to handlers, to commands—is explained in detail. You will learn how to access machine-language subroutines from BASIC to increase the speed of your BASIC adventures. You will learn a variety of ways to access arrays and to store information. You will also learn—painlessly—the logical discipline of structured programming.

Everything you learn from *Writing BASIC Adventure Programs* will increase your skill as an adventure writer—and inspire your imagination.

Other TAB Books by the author

No. 1141 *How to build Your Own Working Robot Pet*

Chapter 1

Adventure Beneath the Keyboard

Board game manufacturers made an unusual discovery some years ago. They discovered that there was a market for complex role-playing games. Not content merely to manipulate a token around a board, players wanted games they could "step into" to exercise their imaginative and strategic skills. A new breed of board games resulted, and a player can now do anything from recreating historic battles to fighting dragons and underworld armies.

Then the age of home computing was upon us, and the imaginative gaming enthusiasts predicted that an alliance between the two fields would not be far off. Imaginative or simulative gaming is, after all, complex, and there are times in which the sheer logistics of playing the game hinder the effectiveness of the simulation. But, if a microcomputer could be used to keep score, manipulate parts, describe situations . . . why, the player could *play* instead of work. It seemed an ideal union.

This sort of union has taken place, but the direction it has taken has been shaped by one other factor: the existence of imaginative gaming programs for larger computers. It was quite a few years ago that Crowther and Woods first cranked out their Adventure program, an amusing simulation placing the player in a danger-filled cavern, fighting troll and snake, dragon and dwarf, as he searches for treasure. This prototype adventure was written for the PDP-10 and was a popular pastime on university campuses long before its little-cousin microcomputer version became available.

Now you can leaf through popular computing magazines and find many adventure and fantasy simulation programs for the home computer—some like the original Adventure, others with new twists. Some are in BASIC, since most home computers at present have ROM-resident BASIC; others are quick, efficient machine-language works.

It occurred to me, however, that no one has taken the time to explain how these various adventure programs work—the programming aspect, that is—and how you might approach the task of constructing one yourself. It is this consideration that this book studies. To simplify the teaching process this book deals with writing an adventure program in BASIC, as opposed to assembly language. This should hold your interest, since all sorts of inventive maneuvers become necessary to make bulky BASIC perform efficiently enough for such a complex type of program.

In this book I use the generic term "adventure" program to refer to any program having the same general play structure and objectives as Crowther and Woods' original offering. My explanations and examples are not taken from the actual code of any commercially available programs. Rather, as you will see, a whole new game program, Basements and Beasties, has been written specifically for this book. My aim is to teach programming skills, not to prevent other hardworking programmers from selling their fine creations.

As an additional example, a second adventure program, called Mazies and Crazies, can be found in Chapters 11 through 13. This program is a somewhat different brand of game, making full use of the TRS-80 graphics capabilities to produce a real-time adventure experience.

WHAT IS AN ADVENTURE PROGRAM?

In its simplest form an adventure program is like a travel folder or a very descriptive map. The player is dropped into a scenario, such as a gloomy dungeon, a steamy rain-forest, or a haunted mansion, and is allowed to move about. The computer describes, in a short paragraph, what the space looks like. (Some recent programs actually draw a map, but let's keep it simple.) The size of the scenario in terms of number of rooms or locations is limited primarily by the available memory of the home computer.

In addition to the descriptive function of the adventure program, the player has a means of communicating with the program to affect his simulated environment. This is accomplished by entering

```
YOU ARE STANDING IN AN OPEN FIELD
THERE IS A JEWEL HERE!
A RAGING BULL PAWS THE GROUND, READY
TO CHARGE!

* TAKE JEWEL
OKAY


* NORTH
HERE IS A SMALL BARN, OLD AND MUSTY


* ENTER
THE DOOR IS LOCKED!
```

Fig. 1-1. Sample run of a hypothetical adventure program.

simple one- or two-word command phrases that are recognized by the program's limited vocabulary (Fig. 1-1). With a handful of phrases the player can move, open and close doors, take objects, or interact with the scenario in other ways.

The goal of the game is for the player to find and keep various articles of worth, "treasures," hidden about the artificial world. He is hindered in his attempt by monstrous creatures, such as trolls, dragons, and spiders—or perhaps more conventional enemies like marauding Huns. The player can die in the fictitious world; usually, he can also be resurrected to continue playing, albeit with a substantial point-loss. The game is not really over until all enemies are vanquished and all treasures are won.

Figure 1-2 gives a sample of the sort of player/computer interchange you can expect in a classic adventure program. Note that the program does not understand all possible inputs, but the cleverer the command interpreter, the better the program.

WHAT YOU NEED?

Adventure programs are one of the last bastions of commercial programming. Home computer enthusiasts have already written their own space war games; with a little instruction adventure programs can also become simple to handle. Relax. You, too, can design computerized labyrinthes and adventures complete with scaly, green things!

First, you'll need a home computer. This book makes the assumption that you own a Radio Shack TRS-80 Model I or III with

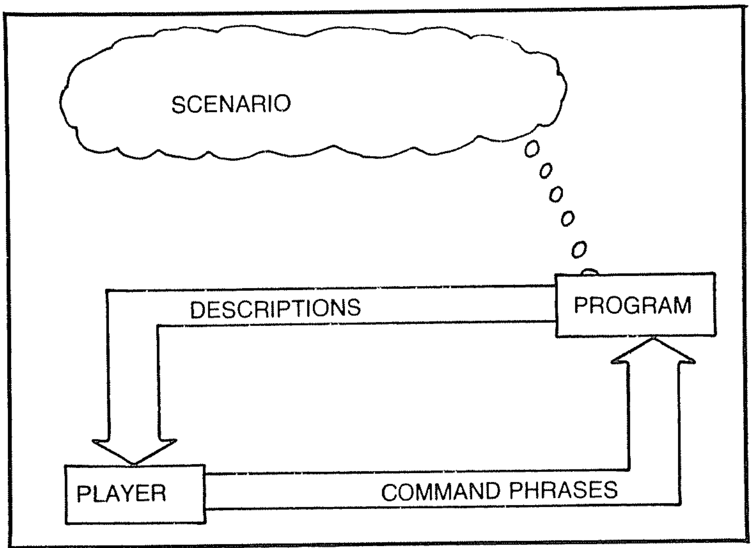


Fig. 1-2. Adventure program /player interaction.

16K bytes of memory. Both of these units operate under Microsoft BASIC, and all programming in this book uses this particular BASIC. However almost any principle in this book can be applied to other BASIC home computers, and the programs themselves should run on other machines with only minor changes.

Also, for storage and recall of the program, a cassette tape (rather than disk) system is assumed. My target reader for this book, obviously, is the TRS-80 owner with a minimal system, the owner that cannot afford all of the extras. How much can be done with only 16K and a tape machine? You'll soon see!

Second, you'll need imagination. You probably have more imagination than you think and simply need to exercise it. Three-quarters of the fun of adventure programming is dreaming up bizarre and unexpected descriptions of the scenario, monsters, and opponents. Read books by J.R.R. Tolkien, Anne McCaffrey, and C.S. Lewis. Read some old mythology and look over some fantasy calendars; you'll be surprised at the ideas you'll get.

Finally, you'll need some good examples. This book provides them. I provide the full listing for a new adventure program called Basements and Beasties. Each chapter describes some detail in the construction of this program and gives some options that you may wish to take. No doubt, this introductory program will play only a foundational part in your own, much more complex adventure program.

WHAT WILL YOU LEARN?

Games are fun, but life is more than just fun and games. So there are some special programming techniques that you'll gain by the end of this exercise; you'll never feel like you are studying in the process!

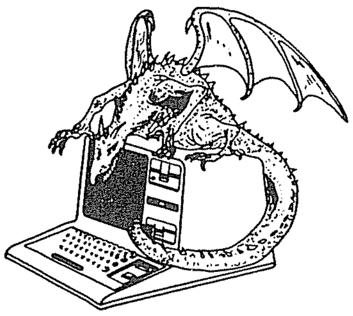
For one thing, you'll learn the wonders of *structured programming*. That sounds formidable, but don't worry. All it means is that you'll experience the joy of knowing where to find a given subroutine in a long program without having to pick the whole program apart line by line. By the end of the book, you'll wish you had written all of your personal programs with some structure. It's easier than you think. (Incidentally, please permit me the use of the term "structured programming" in a much more general, nontechnical sense than is usually meant. Those students of the more formal definition might otherwise wonder if I know what I'm talking about at all!)

You'll also learn many methods of *memory economy*. Adventure programs are, to put it mildly, memory hogs. They eat bytes with long text descriptions, vocabulary lists, and map tables. Remember that the earliest were for big computers. If you have disks and disks to spare, memory is no problem. But, we're writing for a tape-based 16K TRS-80. You'll learn how to conserve and still get what you want.

One final thing you'll learn a lot about is *man/machine interface*. By this I mean how well your program understands inputs from the keyboard and how well it responds. You'll get an education in how to make a simple machine seem far more intelligent that it really is. You'll learn to tailor your program to enhance that link between the scenario and the participant—the sense that the player is really there in that maze, desert, burning fort, or Martian dome.

If by now I have sold you on the benefits of writing an adventure program, then you're ready. Grab a pencil and paper, switch on your computer, and get ready to make the imaginative leap.

Chapter 2



Mapping a Basement Scenario

The primary function of an adventure program is to surround the player with an artificial world, a preprogrammed environment with which he can interact. This substitute environment is effected by a series of textual descriptions and sustained by the presence of objects that can be lying about. This artificial realm is called a *scenario.*

The type of scenario depends on the imagination of the programmer. The original, classic scenario is the underworld cavern environment, in which the player fights mythical beasts to obtain treasures—a medieval land of magic, swordplay, and stone. The sample program in this book, Basements and Beasties, makes use of such a scenario. There are many other possible scenarios, as traditional or as bizarre as the programmer cares to make them. It all depends on your ability to write creatively; if you can describe it, it can be a scenario.

For instance, consider the following possible scenarios:

- The player is trapped in a haunted mansion. He must find all of the treasures hidden in the musty house, while ghosts and ghouls of various sorts hide behind every door.
- The player is lost in a zoo after closing hours—and all of the animals are loose. He must face hungry lions, muscular apes, and angry ostriches as he searches for various items.
- The player is breaking into a top-secret government installation after dark. He must find a number of confidential documents and not

be caught or killed by the many security devices active on the premises.

● The player has crash-landed on an alien planet. He must avoid attacks from the hostile natives while attempting to locate important pieces from his ship's engine, pieces that were scattered on impact.

As you can see, an adventure scenario may be set just about anywhere—as long as there are three parameters active. The first parameter is the background itself, a large environment with room to move about. The second is a set of objects to pursue and locate as the primary goal of the game. The third is a host of obstacles, both living (such as enemies to fight) and inanimate (such as locked doors) to add to the difficulty of the game.

The first of these three factors must be designed before work on the program can progress very far; this chapter deals with creation of the basic scenario. The second and third parameters are handled in the next chapter.

The sample scenario I use is the underground cavern of Basements and Beasties. Your first task is to learn how to map the basement.

START WITH ROOMS TO SPARE

A scenario, for example, our basement, requires the illusion of size. This is accomplished by dividing the scenario into individual units called rooms. These may correspond to actual rooms in a building, separate caverns in a labyrinth, or clearings in a forest, depending on the type of scenario. The adventure program associates each of these rooms with a descriptive paragraph and usually with a short name of two or three words for easy identification. The program provides information about what objects or creatures (if any) can be present in the room when the player enters it. The room is defined uniquely by a preprogrammed description of entrances and exits, that is, directions in which the player must move to reach or leave the room.

Once the programmer has the basic idea and sets out to create his scenario, he starts with a set of undescribed, unspecified rooms. Operating in BASIC with only 16K of user RAM, you are severely limited in this regard. For the sake of demonstration, Basements and Beasties consists of only twenty rooms. (Some refined methodology and machine language can more than double this number.) With these blank rooms before him, the programmer begins to weave a web of pathways between them, until every room bears a spatial relationship to the others. In other words, he creates a *map* of the scenario.

This map of pathways is based on the notation of motion most commonly used in adventure programs, and that is *compass-point travel*. When the player wishes to move, he may choose to go north, south, east, or west. He may additionally move in one of the diagonal directions, such as northeast or southwest. Finally, up and down are also possible directions. This provides ten explicit travel choices for the player. In mapping a room's relationship to other rooms, then, the programmer must define what will happen if the player tries each of these directions while in a given room.

Figure 2-1 shows a diagram representing the possible movement between four rooms. Use a large sheet of graph paper to draw your own map. (A standard 20-by-28-inch sheet of desk blotter paper is great for this.) Then, using a compass, or even better, a plastic template available from an office supply store, draw every room on the sheet. They need not bear any relationship to each other this early in the task—they are simply spaced evenly and in several rows. The map for Basements and Beasties, for example, consists of five rows of four rooms each, for 20 rooms total. Leave plenty of margin between rooms to allow for connecting lines to be drawn from circle to circle.

GETTING FROM ROOM TO ROOM

Think a moment about what happens when you move about from room to room. As you stand in a room in your own house, there are basically three things that can happen if you choose to move in a given direction. These are:

- You may end up in another room. If you go north, and there is a north door, you will find yourself in a new room.
- You may move about in the same room. If you are in a large room, you may go north for quite a while and find that you are still in the same room.
- You may go nowhere. If you go east, you may run into a wall. Ordinarily, you cannot go up or down, either.

Now, imagine that for each room you have a table listing all ten possible directions the player could move. (You can do better than imagine; look at Fig. 2-2.) In each empty space in the table fill in the resulting location of the move. That is, the table tells what room the player will be in if he moves in a direction. Such a device can be called a *travel table*.

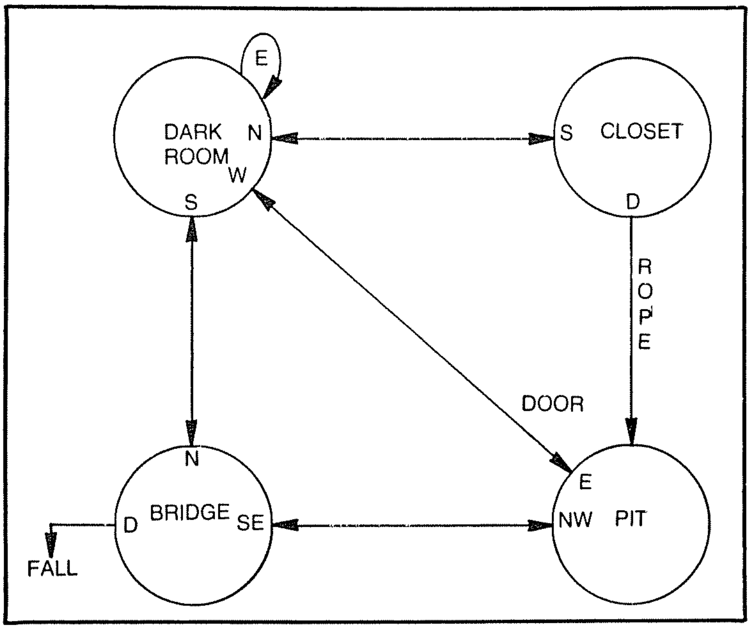


Fig. 2-1. Symbology for the scenario map.

Now, let's look at the three possibilities again. If the player is in room 1, and a north move takes him to room 2, you may write 2 in the space next to NORTH. This is easily represented on your scenario map; simply draw a line from room 1 to room 2 and put an N inside the room 1 circle right at the line. This symbolizes that if the player goes north, he exits the room and ends up in room 2. (Returning works similarly. An S by that line in room 2 indicates that southward travel returns the player to room 1.)

Next, consider option two. In order to give a room the appearance of size, it might be desirable to make certain directions result in no exit at all—simply endless travel. (This is good for outdoor rooms like forests, trackless deserts, etc. When you consider mazes later, this will also be useful.) In such a case, enter the same room number beneath the direction word. If eastward travel leaves a player wandering in room 1, place a 1 in that space. On the scenario map this is designated by a looped arrow, as shown in Fig. 2-1. The E in the center of the looped arrow indicates that eastward motion results in no real progress. The player remains in the same, large room.

Now for the final option. If a player cannot go in a given direction, either because a wall prevents him or because there is no special opening available (in the case of UP and DOWN), this is

TRAVEL TABLE

PRESENT ROOM	NORTH	NE	EAST	SE	SOUTH	SW	WEST	NW	UP	DOWN
ROOM 1	2	2	1	3	0	0	1	1	0	0
ROOM 2	7	9	5	4	1	1	0	0	0	0
ROOM 3	5	0	9	9	0	7	1	1	0	4
ROOM 4	11	0	0	0	5	0	8	2	3	0
ROOM 5	4	6	6	0	3	0	2	0	6	0
ROOM 6	10	0	14	0	0	5	5	7	0	5

Fig. 2-2. Sample travel table with room destinations ordered by attempted direction.

indicated by entering a zero in the respective space. There is no room 0; rather, the zero alerts the program of an attempt to travel in an illegal direction. The program responds by printing a warning message, such as, *"You cannot go that way!"* The player stays in the room as before. In the scenario map this is the unspecified case; that is, any direction not specified by some other symbol is assumed to be an illegal direction with a value of zero in the travel table. In Fig. 2-1, for instance, attempted motion from the closet going north will result in no final motion and a warning message.

There is, in fact, a fourth option, which is a special case. What happens, for example, if a westward direction leads the player off the edge of a precipice and results in his death amid the rocks below? Death is not a destination room, and yet it certainly represents more than endless motion or an illegal direction. Later, you'll see that a special number can be assigned to such a death (a non-room number of some sort, larger than the highest room number in the scenario) as an indicator to the program. When the program encounters that number in the travel table, it knows to assume that the player died (the clod!) and must be resurrected if further play is desired. In Basements and Beasties, for example, the largest room number is 20; so the number 22 represents death by fire, and 23 represents death by falling. There is room for expansion.

CREATING A COMPLEX PUZZLE

Figure 2-3 shows the complete travel table for Basements and Beasties. When you actually write the program in BASIC, this table is resident as a series of data statements, one data statement per line. It would be helpful, then, if you actually created a form like this one, with as many lines as there are rooms in your scenario, and with ten columns for motion.

Wait a moment! Aren't there supposed to be only ten columns, one for each of the ten possible directions of travel? Why is there an eleventh column, marked "Default?" Well, as you will see when you tackle the problem of travel-command input, there are times when a player is not explicit enough about his wishes. What if he types the cryptic "GO IN," or "JUMP?" What direction do they imply? For such inputs you must decide a direction in advance, one of the standard ten, that such an input command implies. This eleventh factor is called the *default direction.*

In the eleventh column you do not put a room number. Rather, you enter a direction number from 0 to 9; wherein 0 refers to NORTH, 1 refers to NORTHEAST, and so on. It is up to you to

PRESENT ROOM	N	NE	E	SE	S	SW	W	NW	U	D	DE-FAULT
1	1	2	2	1	1	1	1	1	0	3	9
2	2	2	2	2	2	1	1	2	0	8	9
3	0	0	4	10	0	0	0	0	1	0	8
4	0	5	0	0	11	0	3	0	0	0	4
5	0	0	0	0	0	4	0	0	0	0	5
6	0	0	0	12	0	0	0	0	0	23	3
7	0	0	0	14	0	0	0	0	0	0	3
8	0	0	0	0	14	0	0	0	2	0	8
9	9	0	16	15	9	0	0	9	0	0	7

10	23	23	23	16	17	17	17	3	0	17	4
11	4	0	0	0	0	0	0	0	0	0	0
12	0	0	13	0	18	0	0	6	0	0	7
13	0	0	0	0	0	0	12	0	0	0	6
14	8	0	0	0	19	0	0	7	0	19	4
15	15	0	15	0	15	16	9	0	0	0	0
16	15	16	16	0	16	0	10	9	0	0	1
17	18	18	18	18	18	18	18	18	18	18	0
18	12	19	0	0	0	0	0	0	0	0	0
19	14	0	0	0	0	18	0	0	14	20	9
20	22	22	22	22	22	22	22	22	19	22	8

Fig. 2-3. Complete travel table for Basements and Beasties.

second-guess the player about the most likely inexplicit command he will enter. For instance, in room 1, there is a hole in the ground. If the player types "JUMP," the most likely default response is DOWN, or direction 9. Much later you'll see how this works in full.

Note that it isn't absolutely necessary to know the description of all of the rooms in the scenario before you weave the web between them. You may find it helpful, however, and if so, you may want to read the section on room descriptions that appears later in this chapter. The present section deals with elements of the web—the structure and relationships that make for an interesting and complex puzzle.

There are four factors that go into a clever scenario labyrinth. These are a home base room, *bottlenecking,* mazes, and obstacles. Let's consider these one by one.

The first room to create for the scenario is known as *home base.* This is usually a room that is separate or outside of the bulk of the scenario. It serves several purposes. The program begins with the player at home base; it is a launching-point for his adventure. Home base is also a place of refuge and a place for safe storage of treasures. The program scores the player on the basis of how many treasures he safely conveys back to home base. This room may correspond, say, to a camp in a hostile jungle, a spaceship in an alien city, or a bathyscaphe in an undersea scenario. In Basements and Beasties home base is a rock pit in which the player, an adventurous archaeologist, has broken through to an underground passage. The home base room is room 1.

Home base represents one access-point to a large closed network of rooms. It is traditional, but by no means necessary, to have at least one more room that provides another access to the bulk of the scenario. Basements and Beasties designates room 2 as an area of ruins with a steel grate set in the rocky floor below—obviously, another doorway into the underworld. A player can travel freely between room 1 and 2, but most motion occurs through the web of rooms underground.

You'll note from the travel table that most of the motion options for rooms 1 and 2 are *looped arrows.* The purpose of this is to produce the illusion of size. A player can go a long way in room 2 and still be lost in the ruins. Only a few specific directions, however, lead to the pit, home-base room 1. This is a good device to use in outdoor situations. Note, too, there are almost no illegal directions (up is the only one), since in a wide-open outdoor environment a person's travel is unrestricted. The zero value finds its use much more naturally in a closed environment, such as a building or a series of caverns.

THE BOTTLENECK PRINCIPLE

The best way to heighten interest in a scenario is to limit the player's options as he moves. That is, he must be made to travel with the maximum effort to see the rooms he desires. One way to do this is by building sections of the web in branches, as shown in Fig. 2-4. Once in room A the player has to choose what avenue to explore. Once the choice is made, the player must return to room A before he can examine a different avenue. This is a good example of *bottlenecking*, forcing a player's motion through a selected room. The end result of several such bottlenecks is a sort of segmented scenario with the ever-present possibility of the player finding new sections previously unexplored.

Another method of bottlenecking is to provide one-way paths. This is a situation in which a player traveling from room A to room B cannot get back to room A by the same path; he is forced to take a more circuitous route. In such a situation the scenario map notation leaves out a return direction on the path line contacting room B. In the travel table a zero takes the place of a return value—or perhaps a different room altogether—as long as no direct return path to room A exists.

Such situations can have various explanations in a scenario description. Perhaps room A is high above room B, and the player

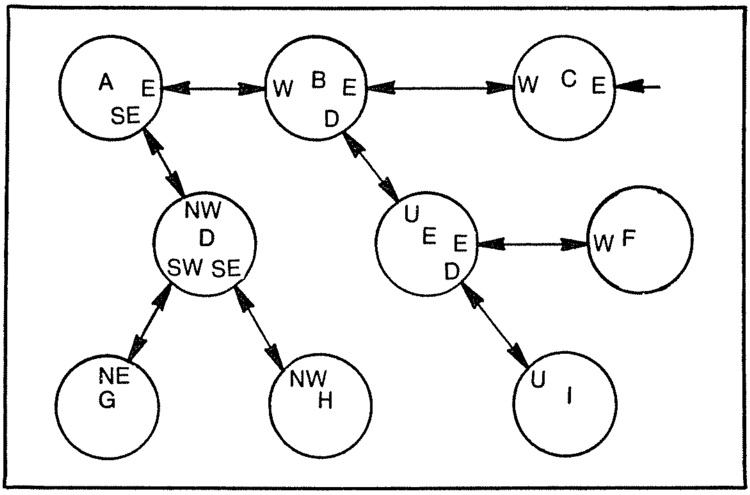


Fig. 2-4. Demonstration of room layout to produce the bottleneck effect.

could jump down to B, but he cannot climb back up. Perhaps rooms A and B are at opposite ends of a department store escalator. Basements and Beasties has a ledge with a stream far below. You can jump into the river and survive; but you'll never get back up to the ledge. You get the picture: one way travel only.

A more subtle sort of bottleneck is a *disguised path*. The strength of this sort of method lies in your ability as a describer. Ordinarily, the room description paragraph tells what doorways or paths are immediately visible around the player. These described doors correspond to the travel table pathways. Nothing says that you have to tell everything, though! Perhaps if the player walks right into those bushes to the north, he'll find a clearing (a new room) that can't be seen. Maybe that south wall isn't as solid as it looks! There may be a waterfall to the east—but, lo and behold, look what happens when you walk right into the water! The idea is to hint at possible secret doors and hope that you'll keep the player in the dark as long as possible.

LOST IN THE MAZE

An adventure program is hardly complete without a maze. Of course, the entire scenario is a maze in one sense of the word, but let's use the term in a narrower sense. An adventure maze is a set of rooms with identical or very similar descriptive paragraphs, such that it is very easy for the player to get hopelessly lost among them. When the player is in such a room, he may choose to move in a direction. If the description of his next room is "YOU ARE LOST IN A MAZE," he wonders, "Am I in a different maze or the same one?" It doesn't take many maze rooms to make an effective mantrap!

Figure 2-5 shows a small three-room maze, identical to the one used in Basements and Beasties. There are three basic elements to a set of maze rooms. I have already stated the first: nondistinctive descriptions. The second is a characteristic that need not be restricted to mazes: it is the use of nonmatching directions in the travel table. Notice, for instance, that a player moves from maze A to maze C by traveling east, but he returns by going northwest, not west. This pattern is carried on throughout the maze, such that the player can never be certain of returning to a given room. You can imagine the frustration, can't you?

The third element is the extensive use of *looped-arrow paths*. Notice that every maze room has three looped paths. The probability is high that a player can make several moves in several directions, thinking that he is visiting entirely new rooms—while in truth he is

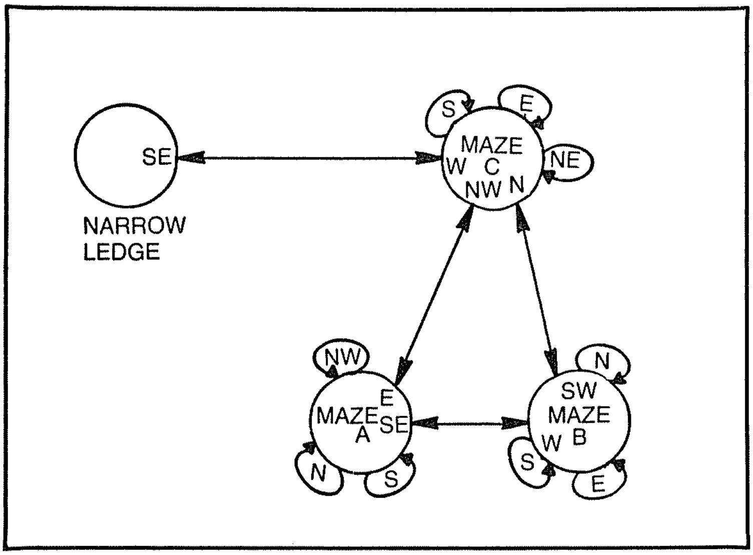


Fig. 2-5. A three-room maze with only one exit.

stuck where he is! The effective use of these three factors can assure player confusion without very many rooms at all.

The standard method used by an experienced adventurer to get out of a maze is to drop an object he is carrying, then move. If the room is described as containing his dropped object, he knows that it is the same room and that he has not yet left it. In this manner, he can effectively tag each room with an object (as long as he has things to drop!) to find his way around. The average adventurer, however, almost never tries this the first time in the maze. Instead, he attempts to run blindly, dashing about randomly until by sheer chance he breaks out!

THOSE ANNOYING OBSTACLES!

Let's take a look at the finished Basements and Beasties map as it looks with all the bottlenecks and flourishes (Fig. 2-6). Ignore the room names for the moment (I'll describe them later) and notice some of the remaining features of the web.

For instance, remember what I said about directions that lead to death, either in fire or by falling. There are three such rooms in "Basements," as you can easily tell from the travel table. Recall that 22 represents a fiery death and 23 represents death by falling. Rooms 6, 10, and 20 have one of these two numbers as a travel-result value. Look, now, at how they are represented on the

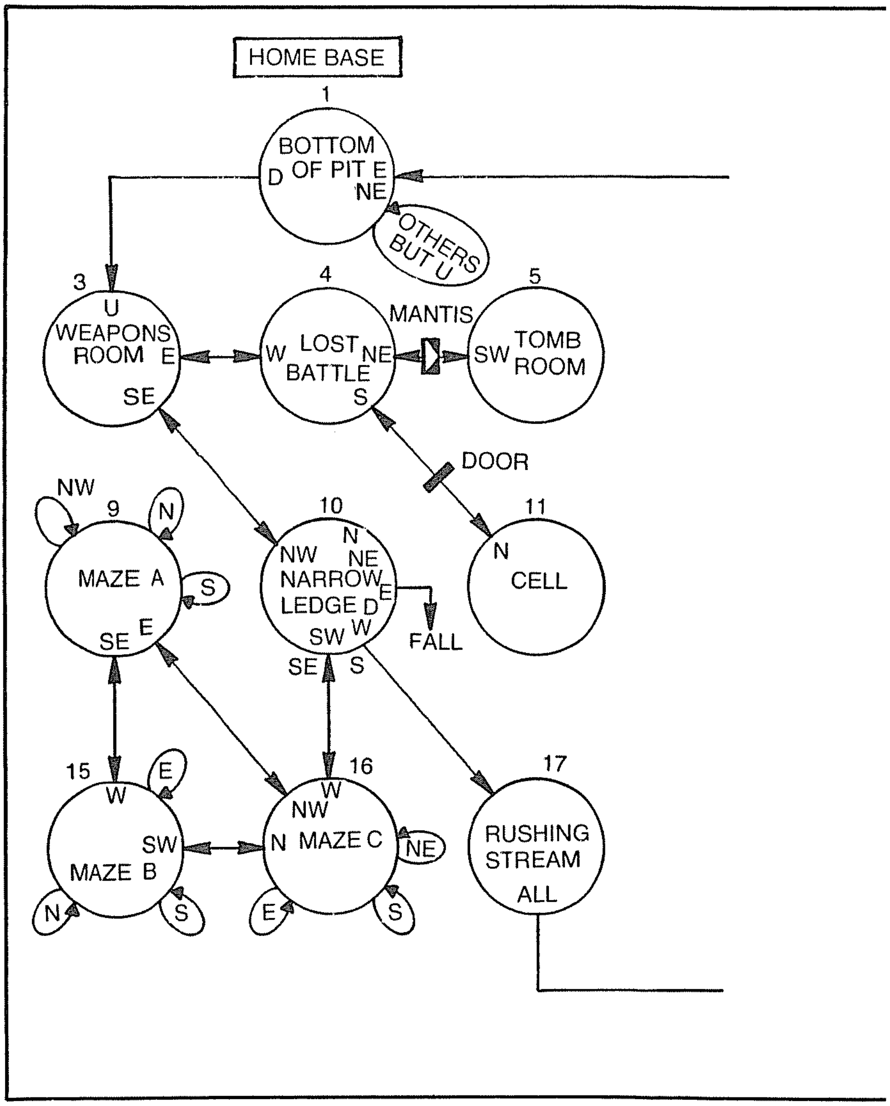
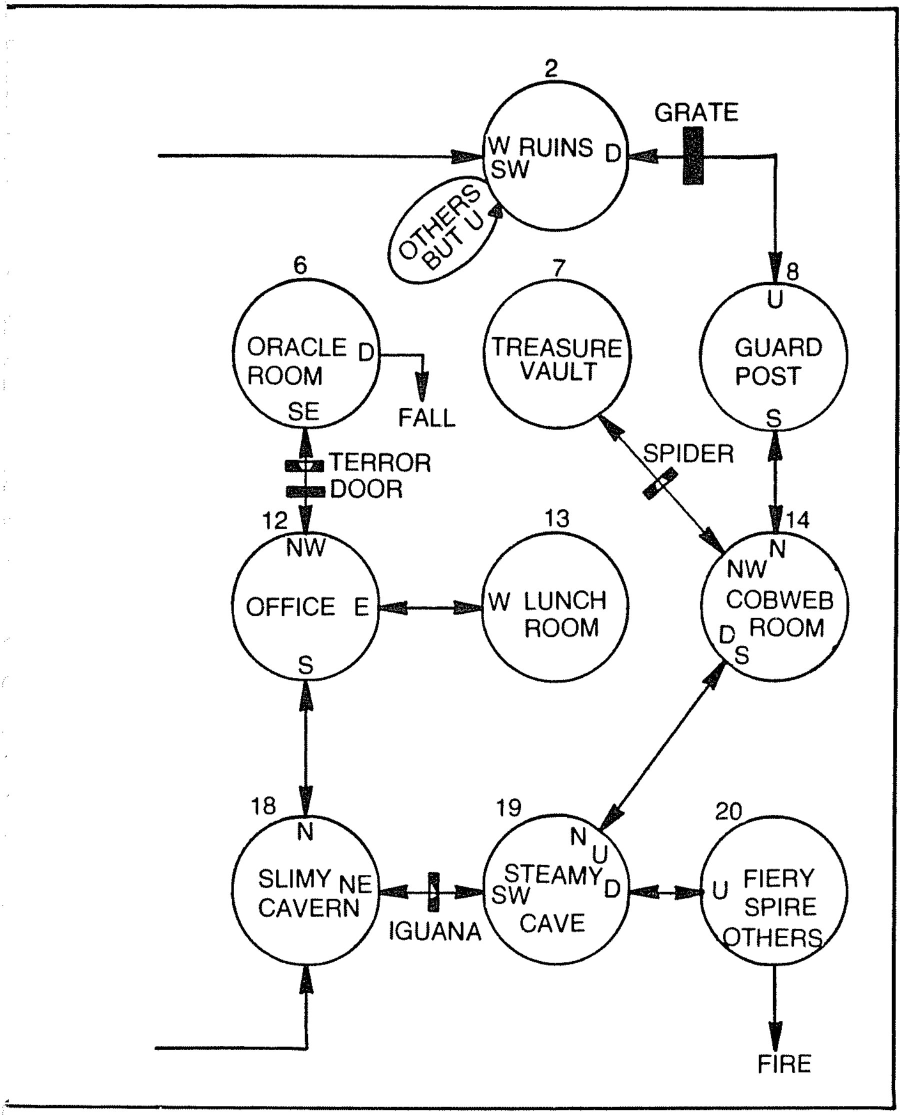


Fig. 2-6. The complete scenario map for Basements and Beasties.

scenario map. A simple arrow marking the travel direction points to a word describing the death: FIRE or FALL.

Realize, please, the manner in which this represents an obstacle. You need not tell the player that a direction is fatal; a hint is enough. Let him make the foolish, presumptious error of testing it. Room 10 is a good example. The description makes both the stream



to the west and the chasm to the east sound equally deadly. If the player tries the stream, he finds a whole different half to the scenario! This tempts him to suppose that perhaps there is more adventure awaiting him in the chasm. He is wrong—dead wrong.

Death is not final in an adventure program; the player can be resurrected, but usually he loses points. He is revived at home base, far from where he died. That means he must make up the distance.

There is, however, an entire class of special objects called *obstacles*, whose sole purpose for existence is to impede the progress of the adventurer. The adventure program maintains a current list of the status of these obstacles, indicating what pathways they obstruct and whether they are still present. This list is, ordinarily enough, called an obstacle list. It is an array of variables maintained by the adventure program in BASIC.

There are two specific types of obstacles. The first type is the living obstacle. These are usually monsters or beasts of some sort that guard a specific room doorway. They usually are overcome by some kind of battle with a given weapon. The second type is the inanimate obstacle. These are things like doors and steel grates. They are removed as obstructions by opening or unlocking with a key. The value of these obstacles as frustrations is increased by the need of special objects—weapons, keys—to deal with them. If, for example, that all-important key is somewhere deep in the scenario, there are several rooms and treasures that the player never sees until he can find the key. Play-time and interest can be extended by such devices.

Let's study the inanimate obstacles first, since they are the most complex to use. On the Basements and Beasties scenario map, notice that there are three such obstacles: a steel grate separates rooms 2 and 8, a door stands between rooms 6 and 12 and another door divides rooms 4 and 11.

In the case of an ordinary door, there are really three possible states. A door or grate may be closed-and-locked, closed-and-unlocked, or open-and-unlocked. It simplifies handling such obstacles if you reduce the number of possibilities to two. Thus, an obstacle is either passable or nonpassable, a simple either-or proposition. The way to visualize this in the scenario is to treat all doors and gates as if they are in one of two possible positions. For programming *A* can represent closed and locked and nonpassable doors; *B* can represent unlocked and freely opened and passable doors.

This simplification affects the realism of the scenario only mildly, but it reduces the complexity of obstacle-handling significantly. If the player tries to go in a direction blocked by a door with a status of *A*, he is prohibited by a message that says, "THE DOOR IS CLOSED AND LOCKED." If he tries to open or unlock it without having a key, the program simply responds, "YOU HAVE NO KEY!" If he does have the key, the program changes the door status to *B* and informs the player by saying, "WITH A CREAK, THE DOOR SWINGS OPEN." Ever after, an attempt to move in that direction is successful—the door is ignored as an obstacle, because its status is B. The player can shut the door and return the obstacle status to A. A message proclaims, "THE DOOR SWINGS SHUT AND THE LOCK CATCHES," and any progress in that direction is subsequently prohibited.

It is obvious that such an obstacle cannot simply be treated as if it were an object sitting in a room. For a door opens onto two rooms; if you open the door while in room A, that same door must now be open if you are standing in adjacent room B. For every inanimate, door-type obstacle then, there needs to be two status numbers: one for each of two rooms affected. The program must simultaneously change the status of both numbers if that door is opened or shut.

CREATING OBSTACLE-LIST ENTRIES

Let's see how the obstacle list and its various entries are created to support the obstacles in an adventure program. Doing so is one of the tougher tricks discussed in this book, so read carefully.

For each door there are two distinct sets of status information that are necessary for the program to properly simulate the obstacle. The following pieces of information are needed:

- What room is affected? This number, naturally, is A for room A and B for room B, the two rooms divided by the door.
- What direction is obstructed? Perhaps the door is on the north wall of room A and the south wall of room B. A direction number of 0 to 9 for each room specifies the direction of motion impeded.
- What kind of obstacle is it? This is primarily to tell the program what messages to print, for example, whether to print "THE DOOR IS OPEN" or "THE GRATE IS OPEN." This number is, of course, the same for both rooms affected.
- Is the obstacle presently passable? This is the actual status indicator that tells whether or not the door is closed and locked. Again, it is always the same for both rooms affected.

Since this obstacle-status data is arranged in a list (the obstacle list), there must be one more piece of information. There must be a number telling the program where the other status information of the two sets is located in the list. After all, once a door is opened, the data needs to be changed not only for the room in which the player stands, but also for the adjacent room. The program can find the status information for the present room easily, but where is the mate to this block of data?

- *Where is the companion status data?* For simplicity, the pair of status blocks are always placed next to each other in the obstacle list.

But, if you're looking at one, is the other one right after it or before it? One simple number can answer that easily by representing "before" or "after." This data is the opposite for the companion block. The first of the pair points to its companion as following, "after;" the second to its mate, "before."

You've just been through a difficult concept, so let's review and simplify a bit. For each door-type obstacle, two rooms are affected. For each of the two rooms, there must be an entry in the obstacle list. This entry must somehow convey five facts: the room number, the direction blocked by the obstacle, the kind of obstacle, the status of the obstacle, and the location of the other entry in the list.

All of these facts can be represented by numbers: two sets of five numbers.

There is fortunately a simpler, more compact way to handle all of these numbers. You can condense all five of these relatively small numbers into one large number. How? Because you can break a large integer into its many digits and allot one or two digits for each piece of data. This is extremely helpful in saving memory space and comes in handy in several places throughout an adventure program.

Figure 2-7 shows the standard format for an integer in Microsoft BASIC. Such an integer has a range of from −32768 to +32767 inclusive. That is, the largest integer you can create has five digits and a sign of plus or minus. As long as the fifth digit is no larger than 3, and as long as you are careful about the fourth digit, you can

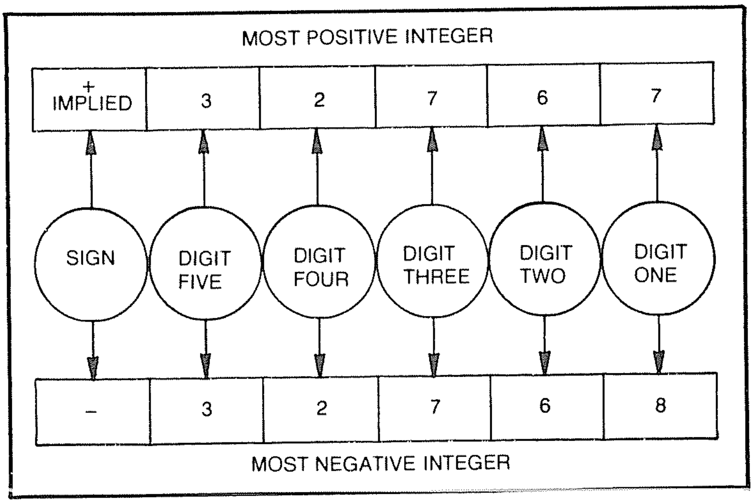


Fig. 2-7. How a standard Microsoft BASIC integer can be broken down for efficient data storage.

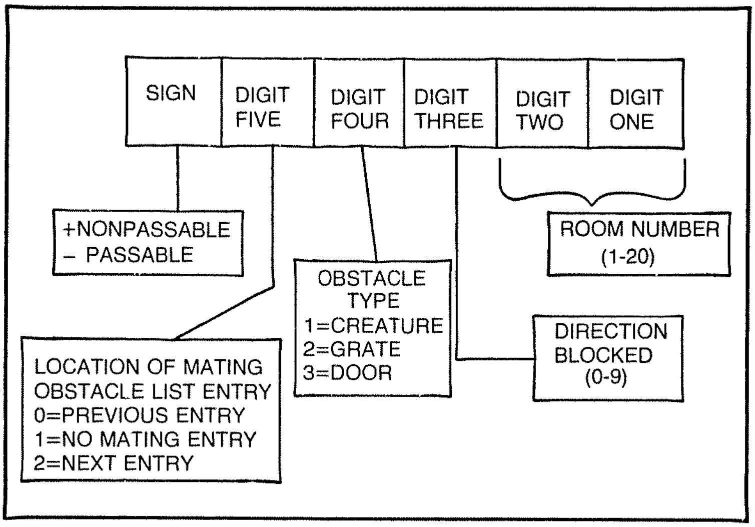


Fig. 2-8. Assigning significance to integer digits to produce an obstacle list entry.

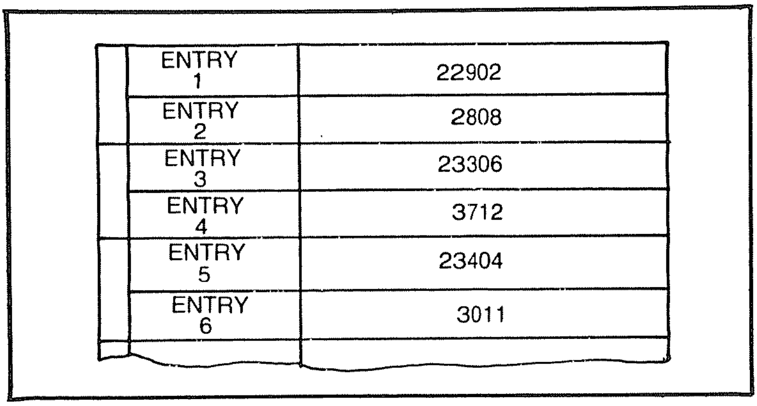
deal with the individual digits and assign any significance or use to them that you need.

Figure 2-8 demonstrates how this method (I call it *integer analysis*) is used to assemble an entry for the obstacle list. First, you need a room number. Room numbers can be as large as 20 (even if there are more in your program, you are not likely to exceed 99). The point is, you'll need two digits of an integer to tell the room number. The first and second digits will do.

Next, you'll need the direction blocked. There are, remember, ten possible directions: the eight compass-points plus up and down. You can assign these numbers of 0 through 9 and plug the data into digit 3. The fourth digit can represent the obstacle type, if you arbitrarily assign numbers from 0 to 9 to them. Digit 5 points to the location of the companion list entry (arbitrarily, a 0 means the other entry is before this one, and 2 means it is after).

Finally, how can you tell if the door is open or closed? Let the sign be the status indicator. If it is positive, the obstacle is nonpassable (closed); if negative, it is passable (open). Useful, right?

The obstacle list—at least as far as inanimate, doorlike obstacles are concerned—consists of a series of integers in pairs, each pair representing a given door. Figure 2-9 shows this portion of the Basements and Beasties obstacle list as it appears when all of the blocks are closed. Take a few moments and try to understand what



ENTRY 1	22902
ENTRY 2	2808
ENTRY 3	23306
ENTRY 4	3712
ENTRY 5	23404
ENTRY 6	3011

Fig. 2-9. The first portion of the obstacle list.

each digit represents. For instance, the first list entry is 22902. You should be able to see that:

- The affected room is room 2.
- The blocked direction is direction 9 (DOWN).
- The obstacle is type 2 (grate).
- The companion-list entry is *after* this one which 2 indicates.
- The grate is closed and locked, which a positive value indicates.

Walk through entry 2 the same way. Remember that if a digit value is zero, it can vanish altogether—at least on surface inspection. In entry 2 the value of the fifth digit is zero; so the entry appears as a four-digit number. Don't worry, think of it as a five-digit number with an invisible, leading zero.

How could an adventure program use the obstacle list to process obstructions? Let's say the player types, "CLOSE DOOR." The program knows two pieces of information: it knows the room where the player is (it always keeps track of this), and it knows that the player wants to close a door, a type 3 obstacle. The program proceeds to hunt through the list, looking for an entry that matches these two criteria. Having found it, the program can set the value of that entry as positive, closing the door for that room. Using the fifth digit the program knows where to look for the second entry it needs to change. Once found, the second entry is also set positive—and a message is printed saying, "THE DOOR SWINGS SHUT AND THE LOCK CATCHES."

There are all sorts of peripheral factors to check, of course. What if the door was already closed? The program could tell by the sign of the list entries and would say, "IT'S ALREADY SHUT!" The list entries are the key to the seeming intelligence of the program.

One more thing needs to be said about the obstacle list. Since the entries require updating from time to time, the list cannot simply be a series of numbers in a DATA line in BASIC. The original values may start off in DATA statements, but these must be loaded into a variable array, so that the individual elements can be set positive or negative as the player interacts with his environment.

BEASTIES AS OBSTACLES

If you've managed to survive this far, you'll have no trouble with the second type of obstacle—the living obstacle. Creatures that inhabit the scenario are much easier to use in the obstacle list.

Consider, first, that doors required two entries in the obstacle list, because in a sense they occupied two rooms at once. Creatures, however, are objects that exist in only one room at a time. Thus (joy of joys!), creatures only require one entry each in the list. There is no need to worry about changing two numbers if a creature's status changes.

Take a look at the scenario map again in Fig. 2-6. You'll soon see that there are four creatures acting as obstacles: one each in rooms 4, 6, 14, and 18. For instance, a player is prevented from traveling northeast in room 4; the Giant Mantis will not let him pass! If the player is in room 5 trying to get to room 4, the Mantis lets him enter, as if the player is sneaking behind him. This reiterates the fact that creature-type obstacles are one-way obstacles only. That's why they only need one entry in the list.

Now, look at the completed obstacle list for Basements and Beasties given in Fig. 2-10. The original three pairs of passive-obstacle entries are there; four new single entries, one for each creature, have been added.

The first creature list entry is 11104. Analyze what it says concerning the obstacle. The block is in room 4. It obstructs motion in direction 1, which happens to be northeast. The obstacle is type 1, which I'll arbitrarily define as "creature." The creature is present, blocking the passageway, since the value of the entry is positive. Now, what about the fifth digit, which was used to point to a second entry? You don't have a second entry with creatures; the fifth digit is set to 1, which means that there is only one entry involved in this obstacle. Later, when you study the actual BASIC code that analyzes list entries, you'll see why the numbers 0, 1, and 2 were chosen for digit 5.

The three other single entries work in the same manner. Searching down through the list, the adventure program can under-

1	22902	STEEL GRATE
2	2808	
3	23306	DOOR
4	3712	
5	23404	DOOR
6	3011	
7	11104	MANTIS
8	11118	IGUANA
9	11714	SPIDER
10	11306	TERROR

Fig. 2-10. Complete obstacle list for Basements and Beasties.

stand that a given single entry represents a specific creature in a specific room guarding a specific pathway. It cannot tell whether it is a mantis or a spider from this list; that distinction is handled by a different list, which you'll soon see. At this point it is helpful just to be able to register this much with a short series of numbers.

How does a creature-type obstacle change status? That is, how does it become either passable or nonpassable? The standard means for this change is battle. If the player has the proper weapon and the random factors of the fight go well, he slays the beast, and the obstacle is resolved. If he fails, the creature continues to guard the path. I don't deal with how battles are fought until Chapter 7, but the part of the program that decides the outcome of the battle is responsible for setting the list entry to negative, indicating a passable obstacle.

Both living and inanimate obstacles are effective in promoting the realism of the adventure scenario. You should appreciate, though, that this luxury is not purchased cheaply. Now you have a variable array that needs maintenance. In fact, every time the player tries to move, the obstacle list must be consulted to see if the attempted direction is blocked or not.

An important consideration of programming that is active in any program of complexity is the trade-off between features and speed. Adding obstacles to a scenario is possible, but the programmer pays for it in processing time. If you're clever, you can keep this handling delay to a minimum.

LOCATING OBJECTS

An adventure scenario consists of more than just rooms, pathways, and doors. There are objects to find as the player wanders through the artificial environment you paint about him. Objects vary in their uses, though. A key that opens a door (and thus renders an obstacle passable) is one sort of object. A sword that slays a certain beast (thus rendering a completely different obstacle passable is another. There are also treasures and other incidentals, such as a lamp or torch to light the way in a dark basement. I've even hinted that creatures are more than obstacles—they are objects in their own right.

In an adventure program there are two factors that define an object: its existence at a specified location (a given room), and its usefulness under special circumstances.

The second of these two factors is wholly arbitrary and is up to the programmer. When a programmer writes a routine that handles, say, the opening of doors, he knows that he has to designate one object as a key. He may choose to treat object 11 as a key. The only thing that makes it a key is the way the open-door routine works. The programmer writes it so that it looks for the presence of object 11 before it can open a door. This goes for other types of objects as well. The only thing that makes object 4 a treasure is that the scoring routine looks for that object number and awards high points for finding it. Object usefulness is flexible.

The location of an object is simpler to handle. In an adventure program the writer simply sets up a variable array in BASIC, assigning one variable for each object in the scenario. Then he merely sets each value to the room number corresponding to its location. If object 10 is in room 6, the array variable for that object is set at 6. For instance, if the variable for object 3 equals 18, it means that object 3 is lying on the floor in room 18. Simple!

Most objects in a given scenario are made to be found and used by the player. That is, each object starts off at an initial location, a given room. When the player enters that room, the program tells him that the object is lying there. He may choose to leave it there, or he may use a TAKE command to pick the object up and put it in his hypothetical carrying sack. In this way he can cart treasures out of the basement and back to home base where he is awarded points for them. (There is, usually, a programmed limit to how many objects can be carried in the sack at one time.)

This raises a good question. When the object, say, a key, is on the floor, it is in the room. Where is the key when the player is carrying it? That is, what value is placed in the array variable for that object? This ambiguity is answered by assigning a fictitious room number to the player's carrying sack. Thus, objects are being carried when their location number is the number of the sack. If the player drops the object, its location number is immediately changed to the present room number.

In Basements and Beasties, which has 20 rooms, the unused number 21 is assigned to the player's carrying sack. (You recall that unused numbers 22 and 23 are assigned to violent deaths that result from travel in dangerous directions.) The sack number is a helpful item. If the player wants an inventory of what he is carrying, the program simply searches through the object location array, looking for any object with a location of 21. These objects are listed.

HELPFUL SCENARIO VARIABLES

An adventure program such as Basements and Beasties makes use of several variables and arrays to keep track of things. I just discussed one such array, the array of locations for objects. I refer to this as the object status array. It has 16 elements in it, since the program has 16 distinct objects. In describing a room, the program must always make one pass through the object status array to see if there are any objects present to describe.

There is a sense in which the player, himself, is an object, at least since he can exist at a location and move from room to room. One variable must always be maintained, apart from the previous array, to keep track of the player's present room number. Let's call it the player location variable. This variable is updated chiefly by the player's use of motion commands.

There are a handful of other useful variables that you need to update from time to time. You need a counter to keep track of the number of steps the adventurer has taken, since this can figure into a scoring scheme. There should be a variable to tell how many objects the player is carrying, so that the program can refuse to let him pick up more than he can bear. Another variable should keep track of how many times the player has been killed, again for scoring purposes. And there are a couple of lesser counters that I touch on later.

I already spoke at length about another important array—the obstacle list. The size of this array depends on the scenario map and the types of obstacles created. Reference is made to it every time a motion is attempted.

One last array needs to be described at this juncture. When the player first steps into a room, the program gives a paragraph-long room description to orient the adventurer. After that first visit, however, the player should not be bothered with a long, drawn-out description. Rather, the program should note that he has already been there once and given him a very abbreviated description, a simple room title. (If he wants the long description again, naturally, he can request it.)

For this feature to work some sort of flag needs to be maintained for each and every room, a flag that indicates whether or not that room has been visited before. The variable is a zero until the room is entered; it is then permanently set to a one. This array of variables is the room status array. The existence of this array is also helpful in scoring, which usually includes some points for number of rooms explored.

This simple yes-or-no sort of flag can simply occupy one digit of the integer stored in the element of the array, allowing the other digits to represent other characteristics of the room. Integer analysis makes this sort of expansion possible.

So what do you have? Figure 2-11 shows all of the variables and arrays used by Basements and Beasties, each with a short explana-

VARIABLE	DEFINITION
CT(0)	PLAYER LOCATION VARIABLE
CT(1)	NUMBER OF STEPS TAKEN
CT(2)	NUMBER OF OBJECTS CARRIED
CT(3)	TOTAL DEATHS OF PLAYER
CT(4)	SLAIN ORCS × 25 POINTS
CT(5)	NUMBER TO BE ANALYZED
CT(6) – CT(11)	NUMBER TO BE SYNTHESIZED
CT(12)	ORC APPEARANCE COUNTER
TX$(0) - TX$(1)	ROOM DESCRIPTIONS (TEMP.)
TX$(2) - TX$(3)	INPUT WORDS (TEMP.)
DA(1)	TRAVEL TABLE VECTOR
DA(2)	WORD TABLE VECTOR
DA(3)	MESSAGE BLOCK VECTOR
DA(4)	OBJECT DESC. BLOCK VECTOR
DA(5)	ROOM DESC. BLOCK VECTOR
RM(1)-RM(20)	ROOM STATUS ARRAY
OB(1,0)-OB(16,0)	OBJECT STATUS
OB(1,1)-OB(16,1)	OBJECT LOCATION
BK(1)-BK(10)	OBSTACLE LIST

Fig. 2-11. Array variables used in Basements and Beasties.

tion. Some of the lesser variables will be described at length in the chapters to come.

DESCRIBING ROOMS AND SCENES

So far we have been speaking of the adventure scenario in very general, structural terms, to convey the methodology or logic behind the idea. When the player enters the scenario, though, he does not see a web of interconnected circles, nor does he see lists and tables. His impression of the scenario is created and sustained entirely by the printed descriptions that are displayed on the computer monitor. These descriptions "paint the picture" in which the player moves and interacts; all of the arrays and tables merely serve as the mechanics to call the proper descriptions.

As the programmer creates a scenario, he starts with a central theme around which to build. I already suggested quite a few—caverns, haunted mansions, office complexes. Once the programmer has an overall environment, the rooms themselves are implied and come easily. The programmer makes a sheet, numbered from one to the maximum number of rooms (in Basements and Beasties it is 1 to 20). Next to the numbers he lists all the different sublocations a player might expect in this type of scenario.

Suppose your scenario is a large office complex. You might list the secretarial pool, the boss' office, the cafeteria, the water cooler, the lavatories. Then there are various halls, the lobby, several offices differentiated by color or size, perhaps a copy-machine room. Once your mind is active, you can probably come up with far more locations than your maximum room count allows! Remember to assign room 1 to your home base, the access point to the larger scenario. In this example, room 1 is probably the lobby, or even the sidewalk or parking lot outside of the building.

Figure 2-12 gives the room list for Basements and Beasties for ease of reference. As you think of rooms, you'll think of little distinguishing characteristics that will be incorporated into the room descriptions later. These may be noted next to the short titles, as shown. The idea is to generate one or two-word identifiers for each room. These are used as the short-form description that is printed for rooms previously visited.

For each room a paragraph description is stored in a DATA statement in the adventure program. The short-form title is also stored along with the larger description. When the player moves into a given room, the program accesses the corresponding DATA statement and retrieves these two pieces of data. Then, depending

#	ROOM	#	ROOM	#	ROOM
1	BOTTOM OF PIT	9	MAZE A	17	RUSHING STREAM
2	RUINS	10	NARROW LEDGE	18	SLIMY CAVERN
3	WEAPONS ROOM	11	CELL	19	STEAMY CAVE
4	LOST BATTLE	12	OFFICE	20	FIERY SPIRE
5	TOMB ROOM	13	LUNCH ROOM	"PSEUDO" ROOM NUMBERS	
6	ORACLE ROOM	14	COBWEB ROOM	21	CARRY-SACK
7	TREASURE VAULT	15	MAZE B	22	DEATH BY FIRE
8	GUARD POST	16	MAZE C	23	DEATH BY FALLING

Fig. 2-12. Room list for Basements and Beasties.

on whether or not the player has seen the room before, the program displays either the long or short description. There is also a LOOK command that specifically requires a reprinting of the long description. After the room list is created, the next test is to write the long description paragraphs for each room.

There are five rules or guidelines concerning the writing of room descriptions. The first is their special format. The entire text of the paragraph must fit on one BASIC program line, sharing that line with the BASIC instruction DATA and also the short title, with a separating comma. This limits the length of the paragraph to about 240 characters, or just over three and one-half lines of text on the TRS-80 monitor. If any commas or semicolons form a part of the paragraph, the whole paragraph must moreover be enclosed in quotes, to prevent BASIC from misinterpreting the DATA statement. The paragraph may also include line-feed characters (which the programmer inserts by typing a shifted down-arrow) to improve the appearance of the paragraph and prevent words from being split in two between lines of text.

The second guideline is the inclusion of *pathway hints*. It is only fair for the description to say, explicitly, "THERE IS A DOOR TO THE NORTH AND A HALL LEADS EAST," in most cases. If no such hints are given, the player is forced to try all ten possible directions to find exits. Note, though, that not all exits need be explicitly told; an occasional room description can even say, "THERE ARE MANY DOORS AROUND HERE." Remember the concept of disguised pathways, too. If the north wall is merely a mirage and really is an exit, just say, "THE NORTH WALL SHIMMERS WITH A STRANGE GLOW," and let the player experiment on his own.

A third guideline is the use of nonoriented language. By this, I mean that the programmer must not make any assumptions about how the player entered the room he is now examining. For instance, imagine a room with two entrances: a trap door above and a steel grate in the wall. It is foolish to display, "YOU FALL INTO A DARK SLIMY ROOM." Even if the trap door is the first means of entrance to the room, there is also the grate. What if the player re-enters the room through the grate later on? The description would be inaccurate. Always describe the room as if the player has suddenly, inexplicably, appeared in the midst. Describe most entrances and exits—even the one he most likely just crawled through.

A fourth guideline is to avoid the use of nonexistent objects; that is, objects not supported by the program itself. This is a hard task and may even be impractical at times, but keep it as a goal. You are bound to get into trouble if you describe some object as a part of the scenery that is not found in the object status array. Why? Because if the description says it's there, the player is bound to try and pick it up! If your program does not make allowances for its existence, that pseudo-object fouls up the realism of the simulation by refusing to budge, or even causing the program to crash. If you must include unprogrammed objects in your description, add something to discourage the player from trying to move it. If your office scenario has a water cooler, say, "THE WATER COOLER SITS NEARBY, RIVETED TO THE WALL." At least there is then an explanation, if only a lame one, for the object's refusal to act like an object.

A fifth and final guideline is to use creative description. Much of the realism of the adventure program depends on your flamboyant, misty-eyed story-telling. There are many ways to make a description stick in the player's mind. The use of color, size, and shape to describe a room are all helpful. Is the room cold and clammy, hot and dry, dark and foggy, tainted by magic, smelling of sea weed, dusty and in disarray? The idea is to convey images above and beyond the explicit words you use.

One of my favorite descriptive devices for adventure scenarios is incongruity. That is, I always have a few rooms that don't seem to fit at all with the time-period or the mood of the scenario. (This goes for objects and creatures, too.) For example, Basements and Beasties describes an underground troll kingdom that for the most part sounds mythical—caves, an oracle room, a room for armor—but I threw in an office and a lunch room, just for surprises. Almost anything goes when it comes to holding the player's interest. Why shouldn't your Martian city scenario have a large, red building with a flying fire-engine in it? Why shouldn't your undersea Atlantis scenario have a shower stall that sprays air? Why shouldn't your old-West scenario have a corner horse-feed station with pumps that dispense "regular" and "premium" hay? You get the idea.

One final word on room descriptions has to do with mazes. Remember that all rooms in a maze should have identical descriptions, to befuddle the wandering player. This can usually be a short phrase like "YOU ARE LOST IN A MAZE," or "HERE IS A SMALL FEATURELESS ROOM."

In maze rooms the long and short descriptions need to be identical. Why? Well, because you don't want the player to know in which maze room he is. If one room gives him a shorter description, he knows he's been there before. Just remember that even if you call those rooms something different in the room list—maze A, maze B, maze C—when the DATA statements are written, the long and short descriptions must be identical.

DESCRIBING WHAT YOU'VE FOUND

Once the room list is compiled and the room descriptions written, it is time to create the object list with its descriptions. When the player enters a given room, the room description is given. If any objects are lying around a line of description is displayed for each of these. This line of description is the long description of the object. Each object also has a short description (analogously to rooms), which is a one or two-word title. This title is used when the player enters the INVENTORY command to examine the contents of his sack.

Figure 2-13 shows the object list for Basements and Beasties. The list is broken up into three basic divisions, each of which I treat in detail.

The first group of objects to create are treasures. These are the objects of worth, finding them is the primary goal of the adventure. In hypothetical spy adventures these treasures are confidential government documents that must be stolen. In Basements and Beasties they are simply objects of monetary worth, such as one might find in dwarfish halls of stone.

To simplify things it is a good idea to make the treasures as unalike as possible, at least with regard to their names. This is to avoid confusing the adventure program when it tries to determine to which treasure the player may be referring. Use names that sufficiently distinguish between the two. For example, each jewel-type treasure is referred to by a specific type of jewel; the player can't say "TAKE JEWEL," he must specify, "TAKE DIAMOND." Even object 1 is not just a jewel, but a jeweled crown.

Remember the principle of incongruity to make things interesting. It would be perfectly acceptable and fun to have a dwarfish transistor radio as a treasure.

The next group of objects to create are the tools. These are objects that are necessary to overcome the variety of obstacles that hinder the adventurer from recovering the treasures. What kinds of tools you create depend largely on your obstacles and vary from scenario to scenario. In cavern-oriented adventures, such as Basements and Beasties, it is customary to have a lamp or torch, since underground environments usually necessitate that kind of tool. The program is tailor-made to limit the player's subterranean motion based on his possession of the lamp or torch. Without the tool room descriptions are prohibited and a message displayed: "BEWARE! IT IS VERY DARK IN HERE!" Of course, this sort of tool is out of place in an outdoor scenario in daylight conditions.

The key, too, is standard fare, because of door-type, inanimate obstacles. The program refuses to open the door or grate unless the key is in the player's carrying sack. If the programmer really wants to get fancy, he can have two or three kinds of keys, each matching a specific door.

The other tools are weapons, basically. The program is constructed so that certain creatures are destroyed only by one or the other weapon. Depending on the scenario, these may range from shotguns to laser cannons. They may even seem harmless in themselves. A simple bottle of seltzer water may be just the thing to stop a marauding robot creature in its own rust!

The last group of objects to dream up are the creatures. These are the monsters that guard the passageways of dim caves, the fully-human Huns of a barbarian scenario, the Martians, or the secret police. Remember that these beasts are also obstacles, and every one of them should have an entry in the obstacle list. When one of them is overcome by fancy swordplay, the obstacle list entry is changed, and the creature is considered dead. (To simplify things the creature usually vanishes rather than allowing a dead body to remain behind. This is done by changing the location of the creature in the object status array to a zero, which amounts to sudden nonexistence, since there is no Room Zero.)

How many objects of the three kinds may you have? That is limited by memory restraints, since every object requires an object status array variable, a long and short description, and programming to handle the special cases that relate to its function. Too many treasures take the fun out of the search; too many creatures are boring. The number and proportions of objects listed in Fig. 2-13 are probably optimum for a scenario of only 20 rooms.

I've already mentioned that, like rooms, all objects have long and short descriptions—long ones for looking in a room, short ones for inventory listings. What guidelines can you suggest for these?

The short descriptions are only one or two-word names for the objects; these are simple. The long descriptions are usually anywhere from 48 to 96 characters long, which is up to a maximum of a line and one-half of monitor text. Usually they are in the form of "THERE IS A BLANK LYING HERE." Treasures usually have descriptions ending with an exclamation point, such as, "THERE IS

A BEAUTIFUL STATUE HERE, ENCRUSTED WITH EMERALDS!" Creatures, too, usually evoke an exclamation, and the description is even more insistent, as in "A GIANT WOOLLY MAMMOTH STANDS NEARBY, READY TO CHARGE IN FURY!"

The only warning is to avoid any mention in the description of the immediate surroundings, since that may change. Don't say, for instance, "THERE IS A SHINY COIN IN THE CASH REGISTER," because the player may take it and drop it in some other room that has no cash register. This rule may be bent a bit in the case of creatures, since they usually live out their existence in one room only. (Helpfully, a part of the program prevents the player from picking up that fire-breathing dragon and carting him off in his sack!)

Once you've created your object list and written the descriptions, you need to add notations to the list, telling in which room each object starts at the beginning of the game. Notice this column of information next to the object names in Fig. 2-13. Where you choose to place the objects is up to you, but here are a few random suggestions. Tools should be placed where they slow progress down a bit. That is, if your scenario has a key, don't give it to the player right off; put it deep into the scenario, so he has to retrace his steps to use it. Put the treasures behind locked doors and behind angry creatures, but leave a few out, unattended, just to whet the player's appetite.

This series of room numbers telling the starting places of objects will later be committed to memory—RAM—in the form of a DATA statement. At the start of the program an initialization routine simply reads these numbers and stuffs them into the brand-new object status array.

THE UNEXPECTED ENEMY

Before I tie a tail to the present discussion of creating a scenario, there is one more item for the program to support. To explain, consider the fact that the creatures already mentioned are pretty tame and fairly docile. True, they are ferocious enough when attacked, but that's just it—they are passive. The adventurer can walk around in the same room as that giant mantis without fear, as long as he does not attack the beast. Now, what kind of challenge is that?

What the program needs is what I term a *tenacious creature*. Tenacious implies that the creature refuses to leave the adventurer alone. There are three characteristics of such an enemy: it wanders

OBJECT	TYPE	DESCRIPTION	STARTING ROOM NO.
1	TREASURES	CROWN OF JEWELS	4
2		GOLDEN CUBE	7
3		DIAMOND BEETLE	20
4		SILVER BELT	11
5		PLATINUM RING	5
6		POLISHED ONYX	19
7		COIN WORTH MILLIONS	7
8		HOURGLASS	6
9	TOOLS	TORCH	2
10		MAGIC AXE	3
11		KEY	10
12		ENCHANTED GRENADE	12
13	CREATURES	GIANT MANTIS	4
14		HUGE IGUANA	18
15		WHITE SPIDER	14
16		NAMELESS TERROR	6

Fig. 2-13. Complete object list for Basements and Beasties.

freely about the scenario, it attacks without provocation, and it follows the player from room to room.

You can see how formidable an enemy this sort of creature is. It wanders around randomly, until it ends up in the same room as the adventurer. It attacks! The player tries to flee, but to his dismay, the foul creature keeps up with him! The player must conquer or be eaten.

Clearly, the tenacious creature is totally unlike the other, passive, creatures, and it is handled much differently. (Study the specifics of the creation of the dreaded Orc in Chapter 4.)

THE NEARLY FINISHED SCENARIO

As you read this paragraph, congratulate yourself on how far you've come (provided you aren't skipping pages). This chapter contains the meat of adventure programming, from the standpoint of form. The remainder of the book actually deals with taking the concepts of this chapter and coding them into BASIC. If you've been sweating through it all and wondering why you ever thought you could handle this sort of programming, relax—the hard foundational work is over.

Let's briefly review the elements of an adventure scenario. As we list each one, see if you can recall what its purpose and function are. If you're foggy about a couple of them, flip back and review them in detail.

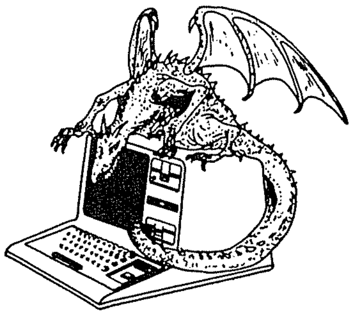
1. A scenario is made up of rooms.
- You need a room list of short room names.
- You need a long description for each room.
- You need a room status array to indicate if a room is unvisited.
- You need a scenario map of room interrelations.
- You need a travel table defining entrances and exits.

2. A scenario is made complex by obstacles.
- You need living obstacles such as creatures.
- You need inanimate obstacles such as locked doors.
- You need an obstacle list defining the obstructions.

3. A scenario is occupied by objects.
- You need treasures, tools, and creatures.
- You need an object list of short object names.
- You need a long description of each object.
- You need an object status array to locate the objects.

Now, at last, you have a feel for much of what it takes to make an adventure program operate. Let's go on now to the next chapter and see how to use this foundation in BASIC programming.

Chapter 3



Structuring the Program

Most programming in BASIC is, sadly, haphazardly done. The programmer starts out with a simple idea, and he writes a simple program. Then, as he adds features to his program, the code grows rapidly and unevenly. At last he is finished, and he has a massive, unwieldy piece of work. The program runs, miraculously enough, but if it needs an improvement here or a correction there, the programmer is stuck. Where is that printer driver subroutine? Where is the routine that updates variables? Lost in a maze of unchecked program growth, the programmer cannot find what he is after.

The writer of an adventure program cannot afford to be sloppy, for at least three reasons. One is memory space. A program like Basements and Beasties needs every byte it can find. Sloppy code is likely to contain redundancies (that could be better organized subroutines) and other items of inefficiency. Speed is another factor. An adventure program tries to do a lot of processing in as short a time as possible. Sloppy code is very difficult to streamline. Modification is the final factor. Someone adventurous enough to write an adventure program will eventually want to upgrade it in some way—extra rooms, new creatures, more treasures. Sloppy code makes program improvement a matter of more frustration than it is worth.

For these reasons, from the very start of your task, do your programming in a very disciplined and thoughtful way. Abide by the rules and partake of the advantages of what is known as structured programming.

HOW TO BE STRUCTURED PAINLESSLY

Nothing sounds quite so ominous as structured programming. In my mind it calls images of lengthy diagrams in obscure notation, reams of flowcharting, and the like. It could put any would-be adventure programmer to flight.

Actually, if you think about it, the rudiment of some sort of program structure is right there in BASIC, staring up from the screen. It is the line number. Think about it for a moment. In Microsoft BASIC the programmer can use any line number from 0 to 65529 inclusive. What usually happens is that the programmer simply numbers his lines as 10, 20, 30, and so on. Then he squeezes extra lines in between if he later needs to add program features. Never does he utilize more than a minute percentage of the numbers available to him.

What does this tell you? Simply this: if you have so many line numbers that you can afford to choose them randomly, you can also afford to choose them meaningfully. That is, you can *assign* certain sets of line numbers to certain tasks. Then, if you ever need to make changes, you know where to check the listing. Instead of scratching your head and mumbling, "Hmmm, I think that routine was on line 639, or 369, or something like that," you can know "That routine was an initialization task; it is somewhere between 0 and 99."

The first key to structured programming is putting the line number to work for you. Use it, as in the example above, to organize your program for easy readability. Later, you'll see how it will help you speed up those important references to DATA statements.

The second key to structured programming has to do with program flow. By this I mean the use of forethought in how a program gets its work done. If you look closely at the flow of your program, you'll see that there are many tasks that it handles similarly, many repetitive paths. When you know what these are, you can write the program so that there are sections of code that serve for many of the program's functions, not just one. That creates an efficient, compact program—just what you need with only 16K of available memory. There are two features of Microsoft BASIC that make this sort of streamlining possible. One is the GOSUB-RETURN feature, which gives you the ability to call subroutines. The other is the ON*X* GOTO feature, the ability to jump to handlers. I take full advantage of these methods in Basements and Beasties.

Now, lest I stray too far into a general treatise on programming, let's get back to your adventure program.

WATCHING THE FLOW

Have you ever seen those drain cleaner ads on TV, the ones with the transparent drainpipes? It is very easy to follow the flow of the plumbing system if you have clear-plastic pipes. Wouldn't it be nice if programming were like that? You could tell what sections of a program get the most work-out.

Figure 3-1 is a transparent-drainpipe illustration of the flow of an adventure program. This sort of diagram is most helpful in dividing the program into logical sections to simplify construction. Let's consider each pipeline and the part it plays.

The first section of the program is the *initialization routine*. This portion of the code is executed only once and serves to set the scenario to some predetermined starting state. What sorts of things are involved in the initialization procedure? For one thing, any string of numeric variable arrays must be created and sized properly. Then those variables need to be set up to simulate the scenario properly. The player-location variable must be set to the home-base room number, for instance.

Several additional blocks of program code are made available to the initialization routine to simplify the process. These are DATA

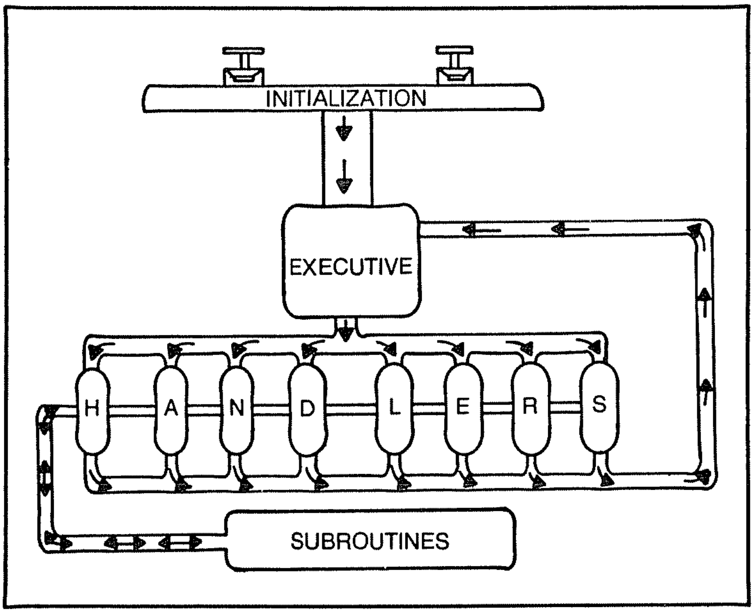


Fig. 3-1. Transparent drainpipe view of the program flow.

statements that contain the values necessary to preset those variables and arrays. One such is the *obstacle initialization block*. The initialization code reads the values from this block and loads them into the array known as the obstacle list (remember, the one that deals with doorways and creatures). A second is the *object initialization block*. This contains all of the starting locations for all objects in the scenario, which are loaded into the object status array. A third block is the *room initialization block*. This is used to fill the room status array, which indicates which rooms have been visited and which remain unvisited.

One question that arises is this: why have a long DATA list to fill the room status array? Don't all rooms start off as unvisited? Can't a simple program loop be used to fill the room status array with zeroes for this purpose? The answer to this question is future expansion. It is conceivable that in the future there may be other status factors you will need to keep track of for rooms. It makes sense to allow for specific values to be loaded into the array, even though in this first version of the program all elements equal zero. Remember that a room status value could be broken into several digits in the future; digit 1 could stand for the visit/nonvisit flag. Other digits could represent other status flags. Accept for the moment that this may prove helpful at some future date.

Now that the scenario has been initialized, the game can begin in earnest. The next section of code is called *Executive*. It is named this because it is the part of the program primarily responsible for the execution of the game; all other parts are subservient to it or eventually loop back to it. (You'll notice how the many other pipelines in the program return to the Executive sooner or later.) The executive has two subsections. One is the *description subsection*. This routine describes the room in which the player stands, including objects and enemies that may be nearby. The other is the *command subsection*. This routine accepts input from the keyboard and interprets the intent of the player.

Two DATA blocks are used by the description subsection, and if your memory is clear, you should be able to tell me which ones. One is the *room description block*. This holds the long and short descriptions for each room, one DATA line per room. The other is the *object description block*, which similarly, has one DATA line per object and holds the long and short descriptions for objects.

There is another DATA block, one that comes into play every time the player enters a command. That is the *word table*. Essentially, it consists of the basic vocabulary of the adventure program, together with corresponding numbers to help define the input word. The command subsection searches through the word table every time a word is entered. If the word is not in the table, the interpreter cannot respond intelligently and must prompt the player for more information.

The next level of code is the row of program units below Executive. These are called *handlers*. There is one handler for each sort of function possible in the adventure program. The command subsection of Executive decides which handler is selected and executed. For instance, if the player types "INVENTORY," the command subsection scans the word table for that word. Upon finding it, the interpreter also reads a number in the table that specifies which handler to execute. Some words are synonymous and invoke the same handler. It is easy to see how the program capabilities can be expanded with this scheme. The programmer simply adds a new key command word to the word table and a handler to perform the new function. The remainder of the code is unchanged, but now the adventure program recognizes a new command. (See how structured programming reduces perspiration in program improvement?)

The particular handlers for motion makes reference to an all-important DATA block which was created in the last chapter. This is the travel table, which contains the entrance and exit information for each and every room. This motion handler and the travel table probably get the most workout of any section of the program code except for the command subsection of the Executive.

Two final divisions of the adventure program serve all handlers and even the Executive. The first of these is a DATA section called the *message block*. For many handlers there are special messages that need to be displayed to indicate the status of that handler. If you walk into a wall, a message says, "YOU CANNOT GO THAT WAY!" If you walk off a cliff, a message says, "YOU FALL TO YOUR DOOM . . .." There are literally dozens of such simple one-liners that must be kept on file for use.

The last division contains all subroutines called by the program. There is a subroutine to locate entries in a DATA block. There is a subroutine to analyze the travel table. There is a subroutine to change the status of an entry in the obstacle list. Many other often-called functions are located in subroutines, all of which reside in a large common pool.

Now let's stand back and look at how these many program divisions are positioned with respect to BASIC line numbering.

PROGRAM STRUCTURE

LINE NUMBERS	PROGRAM SEGMENT
0-99	Initialization
100-199	Main Executive
200-999	Handlers
1000-1999	Subroutines
2000-2999	Room Initialization Block
3000-3999	Object Initialization Block
4000-4999	Obstacle Initialization Block
5000-5999	Travel Table
6000-6999	Word Table
7000-7999	Message Block
8000-8999	Object Description Block
9000-9999	Room Description Block

Fig. 3-2. How to assign specific program segments to specific ranges of line numbers.

Figure 3-2 is the program structure listing for an adventure program. Notice that initialization code (which is executed earliest) is first in the program structure. It may reside anywhere among the line numbers from 0 to 99 inclusive. (I follow the practice of using even line numbers within a block. Basements and Beasties initialization runs on lines 2, 4, 6, 8, and so on.)

Executive is next, then handlers, and subroutines. Notice that more room is given to handlers and subroutines than Executive, since those two sections are more likely to expand. (Of course, you still never come close to using hundreds of available line numbers.)

Next, in multiples of thousands, are the DATA blocks. First are those three blocks used only in the initialization procedure. Then come the two tables for vocabulary and room pathways. Finally, there are the three text blocks, with messages, room, and object descriptions. You'll discover that these text blocks are the real memory-hogs of the program.

GRAB THE DATA AND RUN

Of the above twelve program sections, eight are blocks of data. That means they contain long lists of numbers or words placed on BASIC lines separated by commas and preceded by the DATA keyword.

You may not have realized it before, but getting data out of BASIC DATA statements is rough. There are two BASIC commands that relate to the process. The first is READ. Each time it is executed, one item of DATA is obtained and placed in a desired variable. The next READ instruction gets the next item, until all have been read. The other command is RESTORE, which starts the READ process over again at the very first item of the very first DATA line in the program.

This is all very well and good up to a point. What if, in a DATA list of 300 items, you want item 173? There is only one way to get it in BASIC: do a RESTORE, then run a loop that executes READ commands for 173 iterations until item 173 is read. Now, suppose you want item 160. Can you read backwards to get it? Can you jump to that item somehow? No, you must start all over with a RESTORE and READ, READ, READ until you find it. The problem, then, with BASIC DATA lines is that they require *sequential access*, that is, all items must be read in sequence without skipping any.

You may ask, "So what? I'll just set up a simple FOR-NEXT loop to do all of those useless READs." Fair enough; but consider the problems. The first is time. Your adventure program will muddle along like a turtle if it has to read through all those DATA items sequentially. (Remember that two-thirds of the program consists of data.) Then, too, you'll need to calculate how many loops to do. If you want item 17 in DATA block 4, how many loops do you need? You have to start at the first block and read it, whether you need it or not, thanks to the RESTORE command. To get that item, you must add 17 to the total lengths of the other three blocks. That's work!

One final difficulty is that the data may be numeric in one block and a string in another. If you try to do READ A repetitively, that is, if you try to load the data into numeric variable A, the program crashes if your loop crosses a block with strings of letters in it! Again, the reason is that you can't be choosy in BASIC. You can't skip a block under any circumstances.

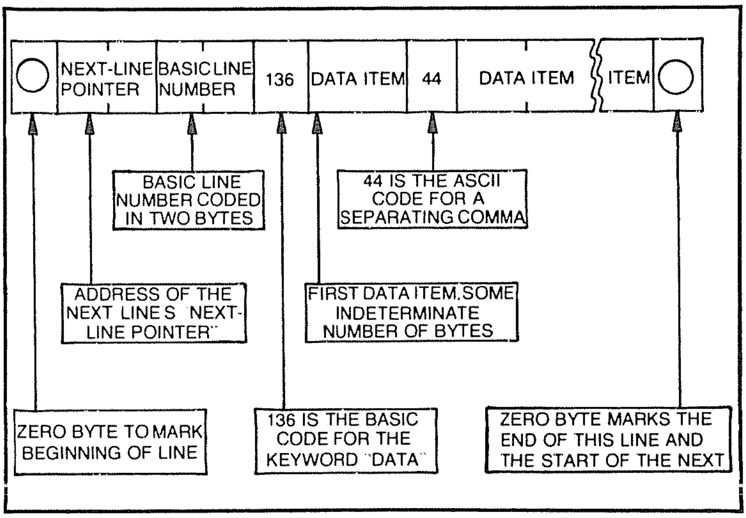
One common way out of this mess is to create a huge variable array and read all of the DATA elements into it. Accessing individual items then becomes easy. The problem with this typical approach is memory space. Essentially the programmer ends up using twice as much memory as the data actually requires! This sort of waste is impractical. It would seem that the limitations of BASIC force us to accept difficult data access or face scandalous memory demands.

Ah, but necessity is the mother of invention. BASIC, after all, is only a program itself, with memory locations that control how it operates. What you need to do is come up with a way to skip around through DATA blocks so you can read what you want to read. Somewhere in RAM memory, BASIC keeps a DATA pointer running that tells the READ command where to read. With a bit of care you can change the value of this pointer to suit your own purposes. To do that, though, you need to understand how DATA statements are stored in memory.

Figure 3-3 shows the format of a DATA statement as it is stored in memory. Notice the following six elements of the format:

- A zero, which separates the DATA statement from the previous statement.
- A next-line pointer, which, coded in two memory locations, gives the memory address of the corresponding nextline pointer of the next BASIC program line.
- A BASIC line number, which is the line number coded in two memory locations.
- A 136, which is the BASIC code for the word DATA
- A list of items, each separated by a comma, which appears in memory as a 44
- A zero, the separator between this line and the next BASIC line, which is just element 1 again

At first some of these numbers in memory take some getting used to, but some simple conventions apply. First, remember that in a 16K TRS-80 all BASIC programs occupy the RAM locations from



O
NEXT-LINE POINTER
BASIC LINE NUMBER
136
DATA ITEM
44
DATA ITEM
ITEM
O
BASIC LINE NUMBER CODED IN TWO BYTES
44 IS THE ASCII CODE FOR A SEPARATING COMMA
ADDRESS OF THE NEXT LINE'S "NEXT-LINE POINTER"
FIRST DATA ITEM. SOME INDETERMINATE NUMBER OF BYTES
ZERO BYTE TO MARK BEGINNING OF LINE
136 IS THE BASIC CODE FOR THE KEYWORD "DATA"
ZERO BYTE MARKS THE END OF THIS LINE AND THE START OF THE NEXT

Fig. 3-3. How a DATA statement with its individual items is stored in memory.

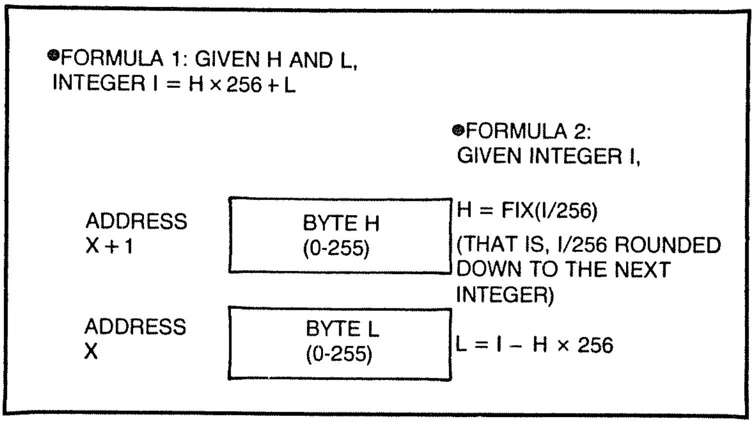


Fig. 3-4. Two-byte code for storing integers.

17384 to roughly 32767. (For owners of the Model I this starting location is 17128.) The content of 17384 is a zero and corresponds to the first element above. It indicates that a BASIC program line follows.

Now, the next-line pointer is in a two-memory-location code, as shown in Fig. 3-4. To calculate the address, multiply the contents of the second memory location by 256, and add the result to the contents of the first memory location. Using this number (which is, of course, between 17384 and 32767), BASIC can tell where each successive line is located in memory. The next-line pointer at the beginning of line A gives the memory location of the pointer in subsequent line B, and so on. The next-line pointer in the very last line of a program is set equal to zero, as a flag to indicate the end of program.

When BASIC is first told to read through a series of DATA statements, it sets a data pointer to the address of the zero that precedes the very first DATA line. Each time a READ statement is executed, this pointer is moved forward, past the piece of data just read, to the comma before the next piece of data. When the last piece of data in that line is read, the pointer points to the zero that marks the end of the line. The next READ causes the pointer to advance, searching for another DATA line and a comma to stop. All DATA lines are read in this manner, until no more data remains. Then, any attempt to read causes an error message. The data pointer in memory always points either to a comma in a DATA line or to a zero preceding a line.

The TRS-80 data pointer is kept in the two memory locations 16639 and 16640, encoded as shown in Fig. 3-4. If you multiply the second number by 256 and add the first, you'll get the memory address of either a comma or a zero preceding a line. The RESTORE command resets the pointer to 17384, the zero at the very start of the BASIC program.

If you know the memory address of the zero that precedes a certain DATA line, you can put that memory address, in coded form, into the data pointer. In this case, a READ statement does not start at the beginning of the program; the program starts with the first item in that DATA line—no matter where it is. Imagine that! Just by changing the data pointer you can begin reading anywhere, skipping hundreds of items if you wish!

The problem is how do you find out what these addresses are? You have eight blocks of DATA statements. The first three are for initialization and are read only once, but the last five are more important. How can the program find the beginning of them?

Here is another benefit of structured programming. You know the line numbers of the five important blocks. As you have seen, each BASIC line contains its own line number in encoded form. What you need is a routine to be placed in the initialization section that does the following:

- Find one at a time the first lines of each DATA block, i.e., 5000, 6000, 7000, 8000 and 9000
- Store these five all-important addresses in a numeric array for future reference.

After this phase of initialization is completed, if a part of the program needs to access a DATA block, it finds the proper address in the array, subtracts one to point to the zero before the DATA line, converts it into the proper two-byte code, and places it into the data pointer. In Basements and Beasties the array is called DA(*n*), for data access. Since you are concerned with the last five DATA blocks, DA(*n*) has five elements, DA(1) through DA(5), containing the proper pointer addresses.

Figure 3-5 gives the initialization code that creates the DA(*n*) array. Let's step through it command by command and see how it determines the proper addresses.

First, some variables are preset. The variable *P* is used for the memory address itself and is incremented successively to the proper address values. *P* is set to 17385, the address of the next-line pointer for the very first BASIC program line. (Remember, 17384 holds the zero preceding this first line.) The variable *N* is incremented from 1 to 5 to step through the elements of array DA(*n*). It begins at 1.

```
6 P=17385:N=1:FORI=5000TO9000STEP1000

8 IFI=PEEK(P+2)+PEEK(P+3)*256THENDA(N
)=P:N=N+1:NEXTI:GOTO10:ELSEP=PEEK(P)+
PEEK(P+1)*256:IFP=0THENCLS:PRINT"ERRO
R":END:ELSE8
10 CT(0)=1:CT(12)=RND(10)+10:CLS
```

Fig. 3-5. Initialization code that loads array DA(*n*) with the addresses of important DATA blocks.

A loop is then set up, to step the variable *I* from 5000 to 9000 in increments of a thousand. Naturally, *I* corresponds to the line numbers for which you are searching. Remember that the line number for each BASIC line is stored in two-byte code early in the line. In the loop you'll need to convert each such encoded line number you encounter into the standard decimal value; if that value equals *I*, you've found the line.

The first part of line 8 does this. Since *P* always points to the first byte of the next-line pointer, the line number bytes are located at *P*+2 and *P*+3. Using our conversion formula, the values in these locations are reconstructed into the original line number and compared to *I*. If the line is found, the present value of *P* is saved in the array at DA(N); N is incremented so that the next line's address is saved in the next array element. The loop is continued and exited upon completion. (Line 10 is the continuation of the initialization procedure.)

Obviously the first line number this routine encounters is not line 5000. What happens when the line number does not match *I*? In that case the BASIC code following the ELSE is executed. *P* is pointing to the present next-line pointer; now it is actually set to the value of that pointer. The contents of the bytes at *P* and *P*+ 1 are converted into a decimal number, and the result is placed in the variable *P*. Now the search can continue, since *P* points to the next available BASIC line. The routine repeats line 8 over and over again until a line number match occurs.

Let's review a moment. Using variable *P* the routine advances through BASIC line by line using the next-line pointer bytes at the start of each line. It looks at the encoded line number in each line, trying to find line 5000. When it finds it, the value of *P* is saved in DA(1). Then the process repeats for 6000 through 9000. The array

DA(*n*) finally contains five memory addresses with which you can locate your five major DATA blocks.

THE ACCESS SUBROUTINE

This part of the initialization routine does half of the work of data access for you: it provides the location of the very start of each DATA block. Now you need a subroutine that can make use of this information to find specific data items. Each DATA block follows the same basic format: it consists of several DATA lines, and each DATA line has several items. The subroutine must do the following things:

- Find the proper DATA block using the array DA(*n*)
- Find the proper row in the block, and
- Set the DATA pointer to that proper row.

With the DATA pointer set, the main program can then use the READ command to locate the desired item in the row. (It may then have to do a READ loop to skip a few items; but the big skip has been done already without any time-consuming loops.)

These three requirements, then, imply the need for two variables that must be set before the subroutine can do its job: a block number and a row number. The block number is a number from 1 to 5, and the row number ranges from 1 to the maximum number of DATA lines in the selected block.

Figure 3-6 provides the code for a subroutine called Access. (Like all adventure subroutines, it resides in BASIC from lines 1000 to 1999.) The main program calls the subroutine only after setting two variables: the variable *A* is set to the DATA block number, and *B* is set to the row number. After Access is complete, any successive READ commands begin at the *B*th line of DATA block *A*.

Access begins by finding the memory address of the beginning of DATA block *A*, using the numbers stored in array DA(*N*). The variable *P* is set to this address. Remember that this address points to the next-line pointer of the first DATA line of that block.

What if the desired row number stored in *B* is 1, that is, what if it is the first line of the block? If so, *P* already points to the proper line, and the subroutine skips on to set the DATA pointer in line 1042. If not, you must search for the right line. The method used is similar to that in the initialization routine. The next-line pointer is read and placed into variable *P*. Each time this is done a line is skipped and *P* points to the next line. This skipping process is done as a loop from 1 to *B* minus 1; the loop skips the unwanted lines until *P* holds the

```
     NAME:    ACCESS

     TYPE:    SUBROUTINE

     INPUT:   A = DATA BLOCK NUMBER

              B = DATA ROW NUMBER

     OUTPUT:  DATA POINTER IS SET TO

              PRECEDE THAT ROW

1040 P=DA(A):IFB=1THEN1042ELSEFORZ=1TO
B-1:P=PEEK(P)+PEEK(P+1)*256:NEXTZ
1042 P=P-1:POKE16640,FIX(P/256):POKE16
639,P-FIX(P/256)*256:RETURN
```

Fig. 3-6. Subroutine Access.

address of the desired line. In this way the proper row is found rapidly.

Now all that is left is to set the DATA pointer so that the READ statement properly reads that line. You may recall that in normal operation the DATA pointer should point to the zero that precedes a DATA line for the READ statement to start with the first data in the line. Well, *P* already points to the next-line pointer in your DATA line, and the zero marker is just one byte earlier. If you set the DATA pointer to *P* minus 1, it is set just the way Microsoft BASIC ordinarily does it—and READ works. Using the formula for encoding numbers into two-byte code, *P* is converted and stored in memory locations 16639 and 16640, which together form the DATA pointer. That's it.

This Access subroutine really speeds things up. For instance, if you need to know the short description for room 7, the procedure is simple. Room descriptions are in DATA block 5, so you set *A* to 5. The room number corresponds to the row number, so set *B* to 7. Then call Access (GOSUB 1040). When it is finished, execute two READ statements; since both are on the same DATA line one reads the long description, one the short. In this manner you have intelligently accessed data, quickly and efficiently, without the need to read every preceding piece of data.

Since any adventure program is at heart a data storage and modification program, it should come as no surprise that several other subroutines are related more directly to the kind of data located in each DATA block. These subroutines use Access to find what is needed. Access can thus be called (though I'll probably never live it down) a sub-subroutine. This is the secret to any truly complex program: simple routines to do simple tasks, other routines that use several of the simple routines to do larger tasks, and so on upward. The result, as you'll see, is that the Executive, the main program of Basements and Beasties, is really rather short and sweet. Why? Because it delegates the detail-work to layers of subroutines below it, a sort of corporate executive routine.

Now that you have a good way of getting at data, what subroutines use this method? Let's look at a few.

GET THE MESSAGE

One function involving quick data access is the printing of special messages. You have an entire data block, block 3, devoted to messages. Life is easier if you assign these messages numbers; when a message needs to be displayed, Access is used to find the right one.

Enter the subroutine called Mesprt, as in message print. Figure 3-7 gives the BASIC code for this utility, which is located at line 1100 in the subroutine section of the program. Only one piece of information is needed for Mesprt to work: the message number from 1 to the maximum number of messages. Mesprt takes over from there, locating the message (using Access, of course), and printing it on the video screen.

The program that wants a message displayed sets *B* equal to the message number and calls Mesprt (GOSUB 1100). Now, in order to use Access, remember, Mesprt must in turn provide two pieces of information: block number and row number. The row number is easy, since the message number is the row number—each row in message storage holds one message. This number is already in *B*, which is where Access would like it, too. The block number is also no problem. Special messages are located in block 3. So Mesprt sets *A* equal to three, just as Access expects. Mesprt calls Access. Now all Mesprt has to do is a READ statement, and it has the message in hand. The message is read, printed, and that's that.

Using Mesprt really frees the adventure programmer from keeping track of his messages. I have seen such programs that have

```
    NAME:    MESPRT

    TYPE:    SUBROUTINE

    INPUT:   B = MESSAGE NUMBER

    OUTPUT:  MESSAGE IS RETRIEVED AND

             DISPLAYED


1100 A=3:GOSUB1040:READA$:PRINTA$:RETU
RN
```

Fig. 3-7. Subroutine Mesprt.

messages planted all over the place, some repetitively. With Mesprt, the programmer piles his messages in one location, and refers to them by number. This saves work, and as you know, programmers can use all the help they can scrounge!

FOLLOWING THE PLAYER'S MOVES

The most frequently entered commands in an adventure program are motion commands. As he progresses from room to room in the scenario, the player is first and foremost an explorer. The programmer wants these commands to take little time to execute, but a lot goes on when the player tries to move. It takes time to find out which room he'll end up in if he moves in that particular direction. Obviously, that block of data known as the travel table really gets a work-out. With Access subroutine, you can dig out the room numbers you need, but it is handy to have a slightly higher level subroutine, one designed strictly for accessing the travel table in the most efficient manner.

So, create the subroutine in Fig. 3-8, which is dubbed Travec, because it finds travel vectors, which are the end destinations of certain moves. Travec resides on line 1120 in the subroutine area of the program. Its primary purpose is to derive destination data from the travel table, given the present room number and the desired direction of travel.

Remember how the travel table is organized? There is one line of resultant destinations for each room. There are eleven elements on each line, the first ten corresponding to the ten possible directions

```
NAME:    TRAVEC

TYPE:    SUBROUTINE

INPUT:   D = DIRECTION NUMBER

         ATTEMPTED (1 - 11)

OUTPUT:  A = DESTINATION OF ATTEMPTED

         MOVE

1120 B=CT(0):A=1:GOSUB1040:FORY=1TOD:R
EADA:NEXTY:RETURN
```

Fig. 3-8. Subroutine Travec.

of travel, the eleventh for the default direction when an ambiguous motion term is used (ENTER, for instance). To find the destination, Travec must first find the DATA line corresponding to the room where the player is. Then it must read across the line to the element corresponding to the desired direction.

Travec uses Access to get to the data. Access asks for two pieces of information: variable *A* must be the row number and variable *B* must be the DATA block number. In using the travel table, the present room number is the important thing. Row number equals room number in the table, so Travec sets *A* equal to the present room number. (The variable CT(*0*) contains the present location of the player). The travel table is block 1 of the five blocks Access covers; so Travec sets *B* to 1. Then, Access is used by executing the proper GOSUB statement.

When Access is done, Travec knows that it can read the proper line of information, but it needs to know which of 11 elements to locate. For this purpose the program that calls Travec must supply one more variable, *D*, to specify the direction of motion. Motion numbers 1 to 8 correspond to compass-point directions; 9 is up, 10 is down, and 11 is the default motion. With this number in *D*, Travec knows just how far over to read. It sets up a short READ loop, counting from 1 to *D*. When the loop is finished, the destination number is stored in variable *A*.

There are several other subroutines in Basements and Beasties that use Access to get at data. These are discussed in detail in later chapters, as their necessity becomes evident.

SQUEEZING DATA INTO INTEGERS

Organized programmers are never wasteful. They are always searching for neater ways to store information, they are always interested in how to compress and compact and combine data. With your memory limitations (most of the 16K is eaten up by text), you likewise cannot afford to pass up a good method for data compression. If you leave it up to Microsoft BASIC to decide, the creation of numeric variables alone will swallow the last of your memory and hand you a nice, big OM ERROR.

The last element of methodology I need to discuss in program structuring is variable organization. Some forethought in this regard should save a lot of trouble when you finally type RUN and hit ENTER.

Let's get the simple preparations out of the way first. For one thing, the adventure programmer must exercise discipline in choice of variable names as he writes the many parts of his program. If you can't remember what variable you last used, don't simply use another. The end result is that most BASIC code is littered with variables from A to Z, when in many cases just a couple would suffice.

Why is this a problem? Well, every time you introduce a new variable Microsoft BASIC proceeds to set up memory space for it. Three bytes of memory are set aside for every variable name you introduce, just as housekeeping, not including the bytes containing the actual value of the variable. These allocated bytes are unused; they simply sit there, wasted.

A good practice is to keep track, on paper, of which variables you are using and for what functions. Whenever you need to use a variable, force yourself to look back at that list and see if a previously created variable will serve. The best example of this sort of organization is in FOR-NEXT loops. Conceivably, you can limit yourself to a few specific variables, like I, J, and K, whose sole purpose is to be used and reused in loops. Resist the urge to leap from letter to letter.

A second rule of thumb to follow has to do with the type of variable you use. After all, not all variables are created equal. Look at Fig. 3-9 and compare the number of bytes involved. The most efficient of the variable types is the integer; it squeezes a number into that two-byte code I've already been using. The precision of integers is poor, since no fractions are allowed; that's why the single-precision and double-precision variables were created. Ordinarily, though, precision variables are for mathematics-oriented programs. Adventure programs have no real need for hair-splitting precision.

Ah, but if you don't tell BASIC what you want, it'll give you more than you bargained for! Single-precision variables are the default type. That is, unless you specify the kind of variable you want, BASIC assumes you want single-precision. This means an extra couple of bytes per variable—a 40 percent increase in variable storage space! There must be a better way.

Of course, if you want to, you can specify "integer" every time you create a variable, by appending a percent sign (%) to the variable name. This is a nuisance, especially since BASIC provides a quicker means, and one that won't accidentally forget to specify a variable somewhere. It is the DEFINT statement.

By using a DEFINT statement in the initialization section of the program, you can prespecify certain variables as integers. The form Basements and Beasties uses is the widest form of the statement: DEFINT A-Z. This effectively tells Level II to treat all numeric variables that begin with a letter from A to Z (and that is all numeric variables) as integers. Effectively you have handed BASIC a note that says, "We're hurting for space, please economize."

MAKING EVERY DIGIT COUNT

The preceding remarks on variable choice are all based on common sense and are nothing new. Now let's get tricky. You have already seen that an integer can hold quite a bit of information. Integers in Microsoft BASIC range from − 32768 to +32767. The sort of numbers you want to store are seldom larger than 10 and almost always less than 100. It is to your advantage, then, to squeeze as much as you can out of one integer.

That simple integer essentially has six areas of storage that are easily accessible in BASIC. There are five digits, (which I number digit 1 to digit 5 from right to left) and one sign place (which is either plus or minus). There are certain limitations to how we can use these six areas. No digits can be assigned such that the final value of the assembled integer exceeds the limits given here. Thus, digit 5 can never equal 4; it must always be from 0 to 3. Digit 4 can be anything from 0 to 9, as long as digit 5 is less than 3; otherwise, the complete integer may exceed 32767. The programmer in using this sort of data compression can best eliminate such worries by assigning digit 5 to some function in which it never exceeds 2. Otherwise, he needs to pay close heed to other digits.

The sign place only conveys one small piece of information, since it is only in one of two states. Still, this is useful, and *it* doesn't really affect the number that follows it. Plus, as you'll see, the sign

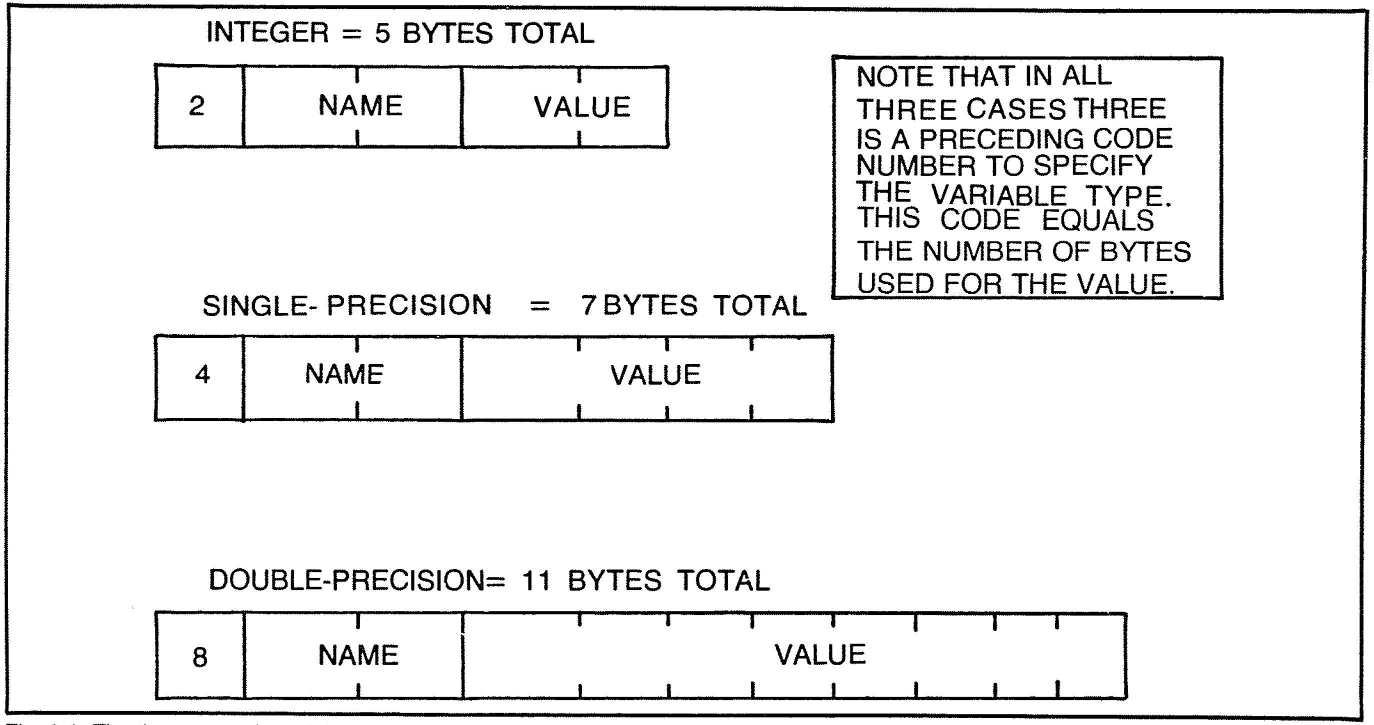

INTEGER = 5 BYTES TOTAL
2
NAME
VALUE
SINGLE- PRECISION = 7 BYTES TOTAL
4
NAME
VALUE
DOUBLE-PRECISION= 11 BYTES TOTAL
8
NAME
VALUE
NOTE THAT IN ALL THREE CASES THREE IS A PRECEDING CODE NUMBER TO SPECIFY THE VARIABLE TYPE. THIS CODE EQUALS THE NUMBER OF BYTES USED FOR THE VALUE.

Fig. 3-9. The three numeric variables and the amount of memory each requires.

place is far easier to test and change than are the individual digits of the integer.

Assuming you wish to use the places of an integer to store small numbers, what methods can you use? There are no BASIC statements that are designed to change the digits of a number directly. So you need to write some routines of your own—two, in fact. One subroutine should take an integer, split it up into five digits, and store each digit value in a separate variable somewhere for easy examination and alteration. The other subroutine should take the values of those five separate variables and reverse the process: assemble them into a complete integer again.

AN INTEGER DIVIDED

The routine that divides a given integer into its digits is called Analyz, and the code for it is shown in Fig. 3-10. The variables CT(5) to CT(11) are dedicated to the integer under examination. When the routine is finished, the first through fifth digits are stored in variables CT(6) to CT(10), respectively. Additionally, the sign of the integer is saved in CT(11); a 1 if positive, a − 1 if negative.

A FOR-NEXT loop is established to clear the values of variables CT(6) to CT(10). This is because digits of some previous integer analysis may remain and confuse the results. Next, the integer must be converted into a form in which the individual digits can be isolated. As a numeric variable, CT(5) cannot be studied digit by digit; no BASIC statement exists to do this. If the contents of CT(5) is converted into a string, the various powerful string-handling statements of Microsoft BASIC can be used to split the string into its components.

The STR$ statement can perform this conversion. In this conversion, the entire number is changed into a string—including the sign! For the moment you simply want to isolate the five digits; a leading sign character would just get in the way. Use the MID$ statement to exclude the first character of the new string. (This first character is a space if the number is positive, a minus if it is negative.)

STR$ converts CT(5) into a string. MID$ creates another string from this one, beginning at the second character. Then, this string is stored in memory as B$. Now a FOR-NEXT loop can be used to analyze B$ on a character-by-character basis. The last character is digit 1, and the digit number increases from right to left. Remember that the number may not have all five digits, depending on its numerical value.

```
NAME:   ANALYZ

TYPE:   SUBROUTINE

INPUT:  CT(5) = A GIVEN INTEGER

OUTPUT: CT(6) TO CT(10) = THE DIGITS

        OF THAT INTEGER, AND CT(11) =

        THE SIGN
1000 FORZ=6TO10:CT(Z)=0:NEXTZ:B$=MID$(
STR$(CT(5)),2):FORZ=1TOLEN(B$):CT(6+LE
N(B$)-Z)=VAL(MID$(B$,Z,1)):NEXTZ:IFCT(
5)<0THENCT(11)=-1:RETURN:ELSECT(11)=1:
RETURN
```

Fig. 3-10. Subroutine Analyz.

The FOR-NEXT loop runs from 1 to the total number of characters in the string: LEN$ determines this limiting value. The string is evaluated from left to right, again using the MID$ statement. As *Z* increments, the MID$ selects each character. VAL does the reverse of the earlier STR$ function; the selected character is converted into a numeric value for storage in a CT(*n*) variable.

The left-hand portion of the equation is designed to ensure that the proper value ends up in the proper variable. For instance, suppose that *Z* equals 1. The digit is the leftmost one in the string. But, what is it, digit 5? digit 4? That all depends on the length of the string; so the LEN statement plays a part. If the number has five digits, the leftmost digit is placed in CT(6+ 5 − 1), or CT(10), the variable for digit 5. This is as it should be.

After the loop has loaded all digits into separate variables, the last remaining task is to store the sign. If CT(5) is less than zero, a − 1 is placed in CT(11); otherwise, a 1 is stored.

The result? Now if a program attributes some significance to say, digit 3 of a stored variable, it simply calls Analyz using a GOSUB 1000 and then examines CT(8). That makes life easier!

AN INTEGER REUNITED

Now, suppose a program used Analyz to check a digit in an integer, and now wants to change that digit. You need a routine to

```
NAME:    SYNTHE

TYPE:    SUBROUTINE

INPUT:   CT(6) = THE DIGITS OF A GIVEN

         INTEGER, AND CT(11) = THE

         SIGN

OUTPUT:  CT(5) = INTEGER

1020 CT(5)=CT(10)*10000+CT(9)*1000+CT(
8)*100+CT(7)*10+CT(6):CT(5)=CT(5)*CT(1
1):RETURN
```

Fig. 3-11. Subroutine Synthe.

will take all of those digits, including the changed one, and reassemble them into a new integer.

This converse of Analyz is called Synthe, and it is shown in Fig. 3-11. It sets CT(5) to the value resulting from the assembly of all five digits in CT(6) to CT(10), even if some of these are only zero. Plus, the sign of the variable is set by the presence of 1 or – 1 in CT(11).

The whole thing can be done much like Analyz, using string-handling functions to convert the digits to string characters, then to concatenate them, then to reconvert the new string to a numeric value. However, the method shown in Fig. 3-11 is quicker and simpler.

After all, each digit really represents a place value in a number. Digit 1 is the ones column, digit 5 is tens, and so forth. So, Synthe multiplies each digit by the proper place factor and adds the results. Then, to set the sign, CT(11) is used as a multiplier. The final result is stored in CT(5), and we have come full circle in integer handling.

AND NOW A STEP DOWN

Well, now we've discussed many of the fine structural points that go into creating a tight, efficient adventure program. It's high time that you opened that creaky trap door and stepped down into the gloom. How are the room descriptions displayed? What about objects? What about attacks from hostile enemies? All of these are part of the main executive section of the adventure program, and all are explained in the next chapter.

Chapter 4

Entering the Basement

I have compared the typical adventure program to a running travelog, providing views of the surrounding environment as the adventurer walks about. The program really has two states of action. The first state is that in which rooms, objects, and the like are described and the program sits dormant, waiting for a command. The second state is that in which a command is entered, a handler is invoked and some sort of result is produced. Ordinarily, the adventure program runs a regular loop between these two states.

Before the first state can be initiated, the program must undergo some preparation. Some of this initialization was described in the previous chapter. Before stepping down into the basement let's complete a look at the preliminaries that allow the program to run.

TYPE RUN AND ENTER

Figure 4-1 shows the entire initialization sequence for Basements and Beasties. When you type, "RUN," and press ENTER these lines set up the ground rules for the execution of the main executive.

First things first. No game program is complete without a snappy title display. It's a shame that you cannot afford to expend precious memory space for helpful things such as rules to the game or playing hints. A title has to do. CHR$(23), of course, places the display into the 32-character mode, producing large attention-

```
2 CLS:PRINTCHR$(23):PRINT@468,"WELCOM
E TO":PRINT@522,"BASEMENTS & BEASTIES
"
4 CLEAR500:DEFINTA-Z:DIMTX$(4),DA(5),
RM(20),OB(16,1),BK(10),CT(12):FORI=1T
O20:READRM(I):NEXT:FORI=1TO16:READOB(
I,1),OB(I,0):NEXT:FORI=1TO10:READBK(I
):NEXT
6 P=17385:N=1:FORI=5000TO9000STEP1000

8 IFI=PEEK(P+2)+PEEK(P+3)*256THENDA(N
)=P:N=N+1:NEXTI:GOTO10:ELSEP=PEEK(P)+
PEEK(P+1)*256:IFP=0THENCLS:PRINT"ERRO
R":END:ELSE8
10 CT(0)=1:CT(12)=RND(10)+10:CLS
```

Fig. 4-1. Initialization code.

getting letters. The PRINT@statements place the title lines just where you want them.

(A note of caution is apropos here for users unfamiliar with the 32-character display mode. The width of the letters is doubled, and every other byte in display memory is skipped. Thus, the PRINT@ statement must be used to address even-numbered screen locations only! For demonstration purposes try to use PRINT@with an odd number; the word is stored in memory, but the screen refuses to display it.)

Next, you need to attend to a number of housekeeping functions within the computer. Some of these have to do with the allocation of memory. Figure 4-1 shows how BASIC line 4 handles these needs.

For instance, you need to tell the TRS-80 how much memory space to set aside for the purpose of constructing and saving strings. You may know that, upon power reset, BASIC goes right ahead and sets aside 50 bytes of space for strings; this space is located in high memory near the memory-size border. You need more than that, though. The printed descriptions for each room have a maximum length of 240 characters, and even the short descriptions used for the objects tend to be at least a line long (64 characters). So line 4 contains the CLEAR 500 statement. This allocates a good 500 bytes of working space for the few string variables used in Basements and Beasties. CLEAR 500 also, of course, resets all variables, a good thing to do as an early part of program initialization.

In the previous chapter I mentioned the need to define numeric variables as integers, in the noble interest of saving bytes. DEFINT *A-Z* alerts the TRS-80 that every numeric variable beginning with a letter from *A* to *Z* (in effect, all such variables) should also be treated as an integer.

Also, any variables that you have chosen to organize into an array must be properly sized or "dimensioned." In the TRS-80, all arrays begin with eleven levels of value in any direction (zero through ten) unless the program specifically indicates otherwise. Thus, if you refer in some line to A(3), BASIC sets up the array A(*n*), where *n* may range from zero through ten. If you never intend to use more than a few of the elements of that array, all of the others represent a memory waste. On the other hand, if you try to refer to something like A(22), the result is a dimension error; you have exceeded the predetermined size of the array.

To save space in the case of small arrays and to make larger arrays possible, the DIM statement is used. Notice that in Fig. 4-1 the single statement DIM is used across five different arrays. In order, the text-string array TX(*n*) is sized, then the data-access array DA(*n*), then the room status array RM(*n*), then the object status array OB(*m, n*), then the obstacle list array BK(10).

The remainder of line 4 performs the initialization of three of these important statuses. Perhaps you recall that the first three data blocks in the program are for the setting up room, object, and obstacle states. Figure 4-2 shows how each of the relevant arrays

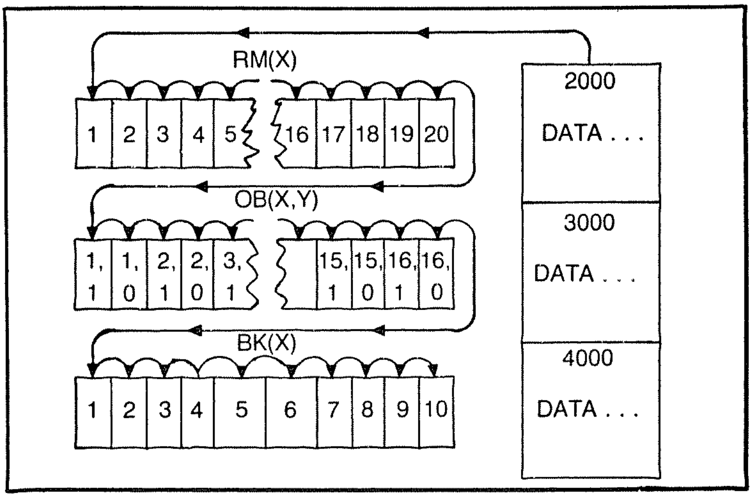


Fig. 4-2. How the major arrays are initialized from the first three DATA blocks.

are loaded from these three blocks. Since the data-read pointer is reset to the very beginning of the BASIC program buffer, upon power reset the first READ statement accesses the first DATA statement, and subsequent READs continue through the blocks that follow. After these initial setups, however, all data access is done using the special methods outlined in the previous chapters: the data-read pointer is controlled by POKE statements within the program, not primarily by BASIC.

The initialization performed in lines 6 and 8 have already been described in the previous chapter. When these two lines are executed, the data-access array DA(*n*) contains the memory addresses of the beginning of each of five important data blocks. These addresses are then used for quick access to the selected data block, skipping previous blocks.

Line 10 is the final bit of preparation you need to begin the game. Array element CT(0) contains the present room location of the intrepid adventurer. The player begins in room 1, the home base; so CT(0) is set to 1. Next, the counter that controls the appearance of the tenacious creature, called Orc, must be initialized. CT(12) is set to some random value of from 10 to 20, using the RND function. At last the screen is cleared, removing the game title and preparing the display for the room descriptions coming along.

DESCRIPTION

Lines 100 to 199, as you may recall, contain Executive, that portion of the code that really gets a workout; most other sections loop back to Executive. Figure 4-3 lists Executive in its entirety. The first two lines, 100 and 102, constitute the description subsection of Executive. They paint the picture of the adventurer's immediate surroundings. Line 104 jumps to a section of code that handles the activity of the tenacious Orc. The remainder of the lines, 105 to 110, are the command subsection. These lines receive input from the keyboard, parse the command, and direct program flow to the appropriate handler. Let's first take a look at the description subsection.

There are three descriptive tasks for this subsection to accomplish:

- Describe the room itself,
- Describe any objects in the room
- Describe a tenacious enemy in the room

Consider first the description of the room itself. There are, of course, two ways to describe a room: the long description and the

```
100 CT(5)=RM(CT(0)):GOSUB1000:C=CT(6):
GOSUB1160:GOSUB1180:IFB=0ANDC=0THENCT(
6)=1:GOSUB1020:RM(CT(0))=CT(5):ELSEIFB
=1THENN=RND(100):IFN<20THENB=5:GOSUB11
00:GOTO580
102 GOSUB1140
104 GOTO112

105 INPUTA$
106 GOSUB1060:A$=TX$(2):GOSUB1080
108 CT(5)=N:GOSUB1000:IFCT(10)=0ORN=0T
HENB=7:GOSUB1100:GOTO104
110 ONCT(6)+CT(7)*10GOTO200,220,240,26
0,280,300,320,340,360,380,400,420,460,
480,500,520,540,560,580,600,620,640,66
0,680,700
```

Fig. 4-3. The Executive, divided into the description and command subsections.

short description. Which descriptive paragraph/phrase should be displayed? The rule is, if this is the first visit to the room, display the long paragraph. On subsequent visits, show the short phrase description. The first piece of information to check, then, is whether this room has been visited before or not.

The room status array, RM(*n*), contains this information. If the first digit of the integer stored in RM(*n*) is a zero, then the room has never been visited; if it is a one, then it has been visited one or more times. You need to check that digit; so the Analyz subroutine, located at line 1000, comes in handy. Analyz divides any integer temporarily placed in variable CT(5) into its five digits, which are stored in variables CT(6) through CT(10). After using Analyz (CT(6) contains the first digit, which would tell you which description to use.

Line 100 begins by setting CT(5) equal to the room status integer for the present room and then calling Analyz. [Note that CT(0) holds the number of the present room; thus, RM(CT(0)) gives the desired status integer.] When Analyz is finished, CT(6) is either a zero or a one, depending on whether the room has been visited or not.

This proves to be a convenient arrangement. The subroutine that actually prints the room description (its name is Viewrm) prints either the long or short form, depending on the following criterion: if the variable C is a zero the long form is used, otherwise the short phrase is used.

CT(6) already meets this requirement (not by chance, I assure you). All you need to do is set C equal to CT(6) and call Viewrm; the proper description will be displayed on the screen.

It's clear that you'll need to take a good look at how the room is described, and so we must take a detour from our analysis of Executive. Figure 4-4 gives the code for Viewrm and one more helpful routine called Darkck. Don't worry, I'll explain how it all hangs together.

Before Viewrm can describe the room, regardless of long or short description, there is one final consideration—is it too dark to see in there? Remember that in Basements and Beasties (and in a number of similar adventure programs) much of the action takes place beneath the earth's surface, in gloomy caves. Standard equipment in such cases is a torch or lantern to see by (that is object 9 in your program). Thus, there are two questions to answer: Is the adventurer in a dark room? Does the adventurer have the torch?

The subroutine Darkck (from DARK Check) evaluates these two questions, which is why Viewrm calls Darkck before it does anything else. Look at line 1180. Using the following logic Darkck sets the variable *B* to a one if the player cannot see his surroundings. If the player doesn't have the torch, and if he's not above ground, then it's too dark to see. The array element OB(9,1) tells where the torch is. If the adventurer is carrying it, OB(9,1) should equal 21, the location number for all things being carried. Then, the only two rooms above the ground and not needing extra light are rooms 1 and 2. If CT(0), the present player location, does not equal 1 or 2, a torch is needed. Darkck uses these comparisons and sets *B* accordingly.

Getting back to Viewrm, Darkck is called. If *B* equals 1 it is too dark to describe the room. In such a case the message "IT IS TOO DARK . . . YOU MAY FALL INTO A PIT!" is displayed in lieu of a description. This message is message 39; all that is needed is to set *B* to this message number and call Mesprt (message-print) at line 1100. The message is displayed and Viewrm returns. (Check the previous chapter on the workings of Mesprt for review.)

If the adventurer can see, though, Viewrm continues on. The long and short descriptions of the rooms are kept in the room description block of data. Using Access (lines 1040 through 1042), the specific long paragraph and shorter phrase descriptions can be read from the DATA line and stored in two separate string variables. The long version is stored in TX$(0), the shorter in TX$(1).

```
NAME:     VIEWRM

TYPE:     SUBROUTINE

INPUT:    C = 0 FOR LONG DESCRIPTION

          C = 1 FOR SHORT DESCRIPTION

OUTPUT:   ROOM IS DESCRIBED IF THERE IS

          ENOUGH LIGHT; IF NOT, A

          WARNING MESSAGE IS DISPLAYED

1160 GOSUB1180:IFB=1THENB=39:GOSUB1100
:RETURN:ELSEA=5:B=CT(0):GOSUB1040:READ
TX$(0),TX$(1):IFC=0THENPRINTTX$(0):RET
URN:ELSEPRINTTX$(1):RETURN
```

```
NAME:     DARKCK

TYPE:     SUBROUTINE

INPUT:    NONE

OUTPUT:   B = 1 IF IT IS TOO DARK TO

          SEE

          B = 0 OTHERWISE

1180 IFOB(9,1)<>21ANDCT(0)<>1ANDCT(0)<
>2THENB=1ELSEB=0
1182 RETURN
```

Fig. 4-4. Subroutines Viewrm and Darkck.

Remember that Access requires two main pieces of information: the block number in variable *A* and the entry number in variable *B*. The number for the room description block is 5, and the entry number is equal to the present room number in CT(0). So Viewrm sets these two variables and calls Access. When Access is done, the data pointer in BASIC is at the beginning of the data line that holds the two descriptions. They are read into TX$(0) and TX$(1) with ease.

The final consideration is which description to use. Ah, way back in Executive we set variable *C* to select the right description! Viewrm just checks the value of *C* and prints out either TX$(0) or TX$(1). That's simple!

Reviewing what we've just seen, Executive needs to describe the room. It calls Viewrm, which may print either a long description or a short one—or it may choose to print no description if the room is too dark.

One more thing needs attention regarding the room. Now that the room has been visited, you need to change the room status array element to reflect the fact. The digits of that element are still kept in variables CT(6) through CT(10). You can simply change CT(6) to 1. Then a call to the subroutine Synthe reassembles the digits and put them into a complete integer in CT(5). (Synthe, described in the previous chapter, is the inverse of Analyz.)

It would not be good to make this change if the room was dark and no description had been printed. Why? Because if the adventurer returns later, torch in hand, he just gets a short description; he was there before, even though he couldn't see. That would be grossly unfair (and adventurers need all the help they can get). So, before you change the room status to visited, ask, "Did he see anything?" That means another call to Darkck, which sets *B* accordingly.

IF *B* is a zero, and *C* (which a long time ago was set to the status of the room) is a zero, change the room to visited. In that case, set CT(6) to a 1 and call Synthe (line 1020). CT(5) is the new room status, and you can place this into RM(CT(0)).

What if the room is dark? In that case, play a little game on the poor adventurer. Remember the message "IT IS PITCH DARK IN HERE . . . YOU MAY FALL INTO A PIT"? Well, provide him with that chance. Using the BASIC RND function to provide a random number from 0 to 100, give the player a 20 percent chance of falling into a pit and being killed by the fall. The variable *N* is set to a random number; if *N* is less than 20, his doom is sealed. Message 5 is printed using Mesprt ("YOU FALL TO YOUR DOOM . . .") and the program jumps out to a handler that takes care of dead adventurers. This may seem cruel and unfair, but it is merely a means to keep smart-alecky players from attempting to travel through the entire scenario without the aid of a torch!

KEEPING TRACK OF OBJECTS

Now that the room has been described, the objects come next. Note that this includes passive creatures that do not attack unless irritated by the adventurer. Executive relies on yet another subroutine for this requirement. Line 102 of Executive calls it.

Figure 4-5 shows line 1140, which is the subroutine Listob (as in list-objects). Its task is to search through the entire object status array, find those objects that are located in the present room, and print their description.

By now you are probably not surprised by the first few statements. It just makes sense once again that if it is too dark to see, no object descriptions can be printed! Here we go again . . . another call to Darkck, and a check to see how variable *B* has been set. Listob returns wordlessly if the environment is too dark.

In the normal case, though, the objects are seen and Listob prepares to describe them. The object descriptions are kept in data block number 4, and Access is used. Variable *A* is set to 4 in expectation of repeated calls to Access. The other variable that Access expects to see, variable *B*, is set by the loop that follows.

In the object status array, the elements OB(*n*, 1) yield the room number where the object is located. Which objects are in the present room? A FOR-NEXT loop is set up for 16 iterations, since there are

```
NAME:    LISTOB

TYPE:    SUBROUTINE

INPUT:   NONE

OUTPUT:  ALL OBJECTS IN THE ROOM ARE

         DESCRIBED IF THERE IS

         ENOUGH LIGHT TO SEE BY


1140 GOSUB1180:IFB=1THENRETURN:ELSEA=4
:FORB=1TO16:IFCT(0)<>OB(B,1)THENNEXTB:
RETURN:ELSEGOSUB1040:READTX$(4):PRINTT
X$(4):NEXTB:RETURN
```

Fig. 4-5. Subroutine Listob.

16 objects. If an object is not in the room indicated by CT(0), then it is skipped. Otherwise, it is a nearby object and needs to be described.

On such objects, a call is made to access to get the object description. The variable *A* has already been set to locate the proper data block. Access now needs the variable *B* to tell it which entry in the block to point to.

Fortunately, we thought to use *B* in the FOR-NEXT loop. Thus, *B* already equals the desired object number, and access has everything it needs to seek the descriptive sentence to be printed. When Access is done, Listob is ready to access the description with a standard BASIC READ statement. The sentence is stored in TX$(4) and is immediately printed. The remainder of line 1140 completes the loop, checking the other objects.

So far the descriptive subsection of Executive has described the room itself and listed any objects sitting around. This also covers the dormant creatures. Now, what about the real fiend of the scenario, the tenacious creature Orc?

ROAMING MONSTERS

The final descriptive task of the Executive is to alert the adventurer to the existence and attacks of the tenacious creature Orc. This creature is unique in that it does not simply pose an obstacle to getting through a given door. Nor does it sit there, refusing to bite until threatened. The tenacious creature Orc as its name implies, never gives up. Once it finds you, it will follow you from room to room, until either you or it is laid to rest. It attacks randomly and just as randomly may succeed in killing the stalwart player. The BASIC code controlling this creature's activity is located in Executive.

Figure 4-6 shows the routine for the tenacious creature, Orc. Three variables are used to control the appearance and activity of the foul beast. Array variable OB(0,1), an unused element in the object status array, is used to store the room location of the Orc. Variable OB(0,0), on the other hand, is a flag. If it equals zero, the Orc has not yet stumbled upon the adventurer. If it is a one, the Orc and the player are in the same room. Finally, variable CT(12) is a counter used to control how often the hero runs into the Orc.

How does the Orc find the player? There are many ways this can be done. For instance, I had one version in which a random number generator bounced the Orc from room to room, until he landed on the player. The problem with this approach, and several others like it, is that it was too random. The Orc might never appear in some rounds; in others, he'd keep popping in every other move!

```
112 IFOB(0,0)=0ANDCT(0)>2THENCT(12)=CT
(12)-1:IFCT(12)<=0THENCT(12)=RND(10)+1
0:OB(0,1)=CT(0):OB(0,0)=1:GOTO116:ELSE
105
114 IFCT(0)<3THENOB(0,0)=0:GOTO105:ELS
EOB(0,1)=CT(0)
116 B=42:GOSUB1100:B=RND(100):IFB>75TH
EN105ELSEB=43:GOSUB1100:B=RND(100):IFB
>60THENB=44:GOSUB1100:GOTO580:ELSE105
```

Fig. 4-6. Routine governing the tenacious creature Orc.

Clearly, he must have limits placed on his random wanderings.

In this version the variable CT(12) is a counter that is set to some random number between 10 and 20. This counter is decremented with every move made by the player. When it runs out, the player meets the Orc! You may choose to change the frequency of meeting, but the concept itself works well.

Let's follow the routine. The first task is to decrement that counter, CT(12). The counter should be decremented under two conditions only. First, the Orc and player should not yet be together, since that is what the counter is preparing for. Second, the player should not be in room 1 or 2, since these are above-ground rooms and Orcs hate the outdoors! OB(0,0) is the flag that satisfies the first qualification, CT(0) the other. If the player is above ground, or if the Orc is with him, the rest of that line is skipped. Otherwise, CT(12) is lessened by one.

Now, what if CT(12) finally runs down to zero? Then the Orc appears! First, CT(12) is reset to some level, to control the next Orc that comes along. Second, the Orc is moved right into the player's room (OB(0,1), the Orc's location, is set equal to CT(0), the hero's location). Then, the Orc chooses whether or not to attack in line 116. If CT(12) has not yet run out, the routine is finished for the time being and returns to the input portion of the program.

If the player is above ground, or if the Orc is with him, line 114 is executed. In the first case, the Orc leaves the player alone if the player moves above ground. Then, OB(0,0) is set to zero, indicating that the Orc is no longer at the hero's throat. (This starts the counter CT(12) back into its downcount for a future meeting.) In the second case the Orc follows the player; so the Orc's location number in OB(0,1) is equated with the player's in CT(0). Line 116 handles possible attacks by the Orc. In line 116 three possibilities are generated: the Orc does not attack, the Orc attacks but does not kill, the Orc attacks and kills the adventurer.

Before these three options are juggled, message 42 is displayed, which warns, "THERE IS AN ANGRY ORC NEARBY!" A random number from 0 to 100 is generated. If this number is greater than 75, the first option above comes true: the Orc does not attack, and the program continues on.

In the 75 percent chance that the Orc does attack, message 43 exclaims, "HE SWINGS OUT AT YOU WITH A BLACK SCIMITAR!" Then the fate of the duel is determined with a second random number. A value of greater than 60 means death for our hero. In that case, message 44 laments, "YOU ARE SLASHED IN PIECES." Then program control skips to a routine that provides handy resurrection and re-entry into the scenario. Otherwise, the program continues on to the input segment.

(I hardly need to tell creative programmers who read this volume that these probabilities are arbitrary. You can demonstrate your capacity either for compassion or cruelty depending on the numbers you choose for the comparisons in line 116!)

With that done, Executive fills a portion of the screen with descriptive material. It now awaits input from the player, who doubtlessly would like to swing his own sword at the Orc before the percentages backfire. The command subsection of Executive now comes into play.

AT YOUR COMMAND

Lines 105 to 110 constitute the command subsection. Through this section, the one or two word phrases entered by the player are broken down and analyzed, and the desired action is performed.

Now, there are far more elegant adventure programs in terms of command parsing (interpretation). Some allow prepositional phrases, adverbs, and so on. Those touches are fine—if you have both the memory and the program speed to handle a large vocabulary and a number of options quickly. You're limited to BASIC and 16K. Don't quail: two-word commands are enough, as long as you choose your vocabulary well.

Figure 4-7 shows the grammar we chose. Every input from the keyboard contains one or two words, either a verb or noun by itself (NORTH or OPEN), or a verb with a noun (TAKE DIAMOND or GO WEST).

It is the place of word 1 to specify the type of task being requested, so that an appropriate routine or handler can be invoked.

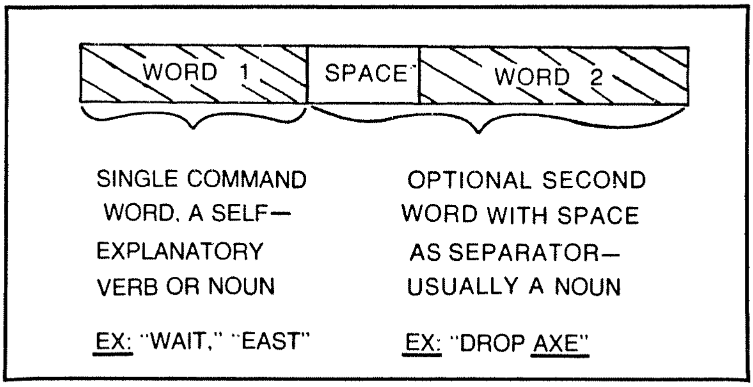

WORD 1
SPACE
WORD 2
SINGLE COMMAND WORD. A SELF—EXPLANATORY VERB OR NOUN
EX: "WAIT," "EAST"
OPTIONAL SECOND WORD WITH SPACE AS SEPARATOR—USUALLY A NOUN
EX: "DROP AXE"

Fig. 4-7. Simplified grammar of Basements and Beasties.

For instance, if the player types in "SCORE," a handler is called that displays the present score and then returns to Executive.

Word 2, on the other hand, specifies the parameters relating to the task implied by the first word. Suppose you typed "TAKE." What does the program do? It invokes the handler called Take, but what object in the room should the adventurer take? The second word removes the ambiguity by supplying additional data.

In either case the words of the command must be recognized to be useful. This involves that dreaded trick of the programmers' trade, the *table search.* A word table must be maintained in memory so that each input word can be compared to the table elements for identification.

The word table for Basements and Beasties comprises data block 2. It is not enough, of course, simply to have a long list of words; each word should have some data associated with it, to instruct the command interpreter on how to define it. Each word in the table is paired with an integer known as the *word ID number.* Each of the digits of this integer contains information to define its accompanying word.

Figure 4-8 gives the breakdown of the word ID number. There are three fields of information that aid in identifying a given word. The first is digit 5; if it is a one, the word is a valid first word term and should be interpreted as such. Any word in the table with an ID number of from 1000 to 19999 invokes a handler, but which handler? The answer is in the field consisting of digits 1 and 2. These specify one of 99 possible handlers that this word can imply. If you enter the word SCORE, it is found in the word table with an ID number of 10012. The command interpreter then knows to invoke handler 12.

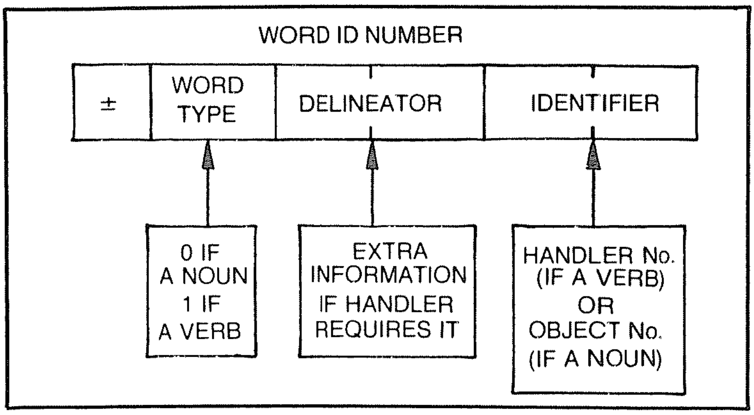

WORD ID NUMBER
±
WORD TYPE
DELINEATOR
IDENTIFIER
0 IF A NOUN 1 IF A VERB
EXTRA INFORMATION IF HANDLER REQUIRES IT
HANDLER No. (IF A VERB) OR OBJECT No. (IF A NOUN)

Fig. 4-8. Digit assignment for the word ID number.

A third field is added, made up of digits 3 and 4. In some special handlers extra information can be carried through this field. The simplest example of this usage is a handler called Liners. This handler simply gives a one-line answer to a one-word input. If the player types "WAIT," he gets the message, "TIME PASSES." Many different words can invoke Liners, but which message should Liners print? To simplify matters the third field in the ID number contains the message number that Liners use for that word. In the word table the word WAIT is paired with the integer 13809. This tells the interpreter to invoke handler 9 (which is Liners). It tells Liners to print message 38 (which is, "TIME PASSES"). There are other handlers that use this third field, too, but you get the idea.

Let's get back to the first field. If it's a one, you have a valid first word; if it's a zero, it's a valid second word. The handler needs some additional information, such as the name of an object to TAKE or a creature to KILL. In the case of a second word identification, the second field (first and second digits) represent the object number to which the word refers. (The third field is not used.) Essentially, object names in the word table are simply paired with their numbers, since all other digits in the ID number are zeroes.

Note, too, that many different words stored in the word table refer to the same object; they all must be paired with identical ID numbers. That is why the words JEWEL and CROWN both are paired with ID numbers of 1, because they both refer to the first object, which is the jeweled crown.

Now it is time to consider the actual code that makes use of the word table and the word ID numbers, and see how the handlers are invoked. Use Fig. 4-3 as a reference to the discussion.

The first step is simple—getting the input string. The BASIC statement INPUT A$ produces a question-mark prompt on the TRS-80 screen and loops until the player types in a series of characters terminated by ENTER. The input string is stored in the variable A$.

Now, think for a moment: where is the first or second word in that variable A$? To the TRS-80 A$ is just a series of characters! There must be a process to break A$ into one or two input words. This process is embodied in the subroutine called Getcom. It resides at line 1060. Figure 4-9 shows its contents.

Given a string in A$, the purpose of Getcom is to isolate the one or two words in it and place these in the variables TX$(2) and TX$(3) as the first and second words, respectively. If there is only one word in A$, it is placed in TX$(2) as the first word, and TX$(3) is nulled to indicate no second word.

The key to this isolation process is the space character. If the input string stored in A$ contains a space, it is assumed that this is the separator between the first and second words. If there is no space, A$ is considered to be one entire first word. Getcom must systematically search through A$, looking for a space.

Fortunately, Microsoft BASIC contains some very helpful string manipulation functions. The LEN(X$) function can determine

```
   NAME:    GETCOM

   TYPE:    SUBROUTINE

   INPUT:   A$ = COMMAND INPUT LINE

   OUTPUT:  TX$(2) = WORD 1

            TX$(3) = WORD 2, IF ANY


1060 FORI=1TOLEN(A$):IFMID$(A$,I,1)<>"
 "THENNEXTI:TX$(3)="":TX$(2)=A$:RETURN
:ELSETX$(3)=MID$(A$,I+1):TX$(2)=LEFT$(
A$,I-1):RETURN
```

Fig. 4-9. Subroutine Getcom.

the length of the input string, so you'll know how far to search. The MID$(X$,y,z) function can extract specific characters out of the string for your examination.

Here's how it's done. A loop is set up to search the string from the first to the last character. Each character in the string is compared to a space. The expression MID(A$,I,1) selects one character from A$, specifically the one that is *I* characters from the beginning of the string. As *I* changes value, each and every character is checked to see if it is a space. Each time a character is found not to be a space, the loop continues.

If the loop runs out without having found a space, TX$(3) is set to a null length, meaning that no second word exists in the string. A$ is interpreted as being only one word, and it is stored in TX$(2) as first word. Getcom is finished and returns.

If a space is found, however, Getcom makes the assumption that all of the characters to the left of the space are word 1; and all of the characters to the right of the space are word 2. First, word 2 is stored in TX$(3); the expression MID$(A$,I+1) extracts all characters from position *I* + 1 to the end of the input string. (Note that this excludes the space itself.) Then, a word 1; is stored in TX$(2). The expression LEFT$(A$,I−1) extracts all characters from the beginning of the input string up to and including position *I* − 1. (Again, the space is excluded.) Getcom's task is finished; so it returns.

You may be asking, "What if there are more than two words in the input string?" Well, think it through. Getcom makes the division at the very first space it can find. It doesn't continue to see if there are more spaces or words. Therefore, if you type in "KILL SPIDER QUICKLY," word 1 is "KILL" and word 2 is "SPIDER QUICKLY." You'll see in a moment that the useless third word is safely ignored when the interpreter figures out what creature is intended by the second word.

Back to the command interpreter itself. After it calls Getcom to divide the input string, it has a first word with which to work. The next thing to do is to find that word in the word table, get the word ID number, isolate the handler number, and invoke the handler. Simple!

The thing that makes it simple is yet another subroutine; this one is called Idword; Figure 4-10 gives the code for it. The interpreter sets A$ equal to the first word and calls Idword. Idword takes the word in A$ and locates the word in the word table. Upon finding the word, it sets the variable *N* equal to the word ID number paired with the word.

```
  NAME:    IDWORD

  TYPE:    SUBROUTINE

  INPUT:   A$ = WORD

  OUTPUT:  N = WORD ID NUMBER IF FOUND

               IN WORD TABLE

               N = 0 OTHERWISE


1080 IFLEN(A$)>5THENA$=LEFT$(A$,5)
1082 A=2:B=1:GOSUB1040
1084 READB$,N:IFB$="."ORB$=A$THENRETUR
NELSE1084
```

Fig. 4-10. Subroutine Idword.

The first step in the process is in line 1080. Essentially, all this line does is limit the word in A$ to a maximum length of five characters. You may have wondered, if you looked at the word table, why all the object names and other terms are all only five letters long. This is strictly to save space. It turns out that five is an optimum length for word recognition in adventure programs; certainly fewer than four letters causes some ambiguities and erroneous identifications. This also allows for a bit of input abbreviation. The player can type "INVEN," and the program knows he is asking for an inventory. Fumble-fingered typists lost in the heat of adventure play always appreciate a break!

Next, Idword gets ready to begin its long reading through the word table. The word table is data block 2, and Idword wants to begin searching at the first entry. It sets variables *A* and *B* accordingly and calls the ever-ready subroutine Access to position the BASIC DATA pointer at the head of the table. Subsequent READ statements access the elements of the word table.

The search performed by Idword is a good, old-fashioned sequential search: the word table is not alphabetically sorted. As it turns out, the time delay involved in finding the word is not too long; so I never wrote a fancy binary search routine. (Chapter 10 of this

this book, however, does provide a way to use an alphabetized word table to speed up the search.)

Sticking with the sequential search, Idword reads data in pairs, grabbing a word into variable B$ and placing its corresponding ID number in variable *N*. Then it performs a compare operation. Obviously, if B$ equals the input word A$, the job is finished and Idword returns with the ID number still in *N*. But another comparison is performed, to see if the character "." has been read from the table. The last elements in the word table always are a period paired with an ID number of zero. Thus, if Idword reads a period in its search, it knows it has reached the end of the table without finding the word it is after. Just the same, it takes no more action; it returns. The variable *N*, though, now contains a zero, which is a reserved ID number indicating a search failure. If the program that called Idword (the interpreter) gets back an *N* with zero value, it knows that the input word is not in its vocabulary, and it can respond accordingly.

Once again, back to the interpreter. Now that it has the variable *N*, the interpreter can begin to break down and make use of *N*.

You naturally remember the subroutine Analyz, which isolates the digits of a given integer. The interpreter places the value of *N* into variable CT(5) and calls Analyz. When that routine is completed, the five digits of the ID word reside in CT(6) to CT(10).

Now the interpreter needs to make a few decisions. What if the player entered a single word that is not really a valid first word, like "SPIDER?" Or what if the player typed two words, but the first is not a valid first word, such as, the phrase "SPIDER KILL?" The interpreter rejects both of these entries by checking the value of the fifth digit in the ID word. That digit must be a 1 to be a valid first word. If it is not, the interpreter plays dumb: it sets variable *B* to message 7 and calls Mesprt. The result is the displayed question, "WHAT DID YOU SAY?" The program-flow loops back to the INPUT A$ statement, allowing a new command input from the player.

At the same time the interpreter checks the value of *N* to see if the input word was recognized from the word table at all. If *N* equals zero, once again the interpreter professes ignorance and prompts the player for another command with the question in message 7, looping back to line 104 and INPUT A$.

You're skeptical; I can hear you again! You are asking why the interpreter is so dumb. After all, it should be smart enough to ignore word order in that input example, "SPIDER KILL." It is obvious to dumb humans what that phrase was intended to mean: why not to a dumb computer? Again, it's a case of personal preference. The command subsection of Executive can be refined to become quite literate and comprehensive, if the programmer is willing to sacrifice some memory space and speed.

If the program gets through these few input constraints, it decides that it is ready to invoke a handler; the first and second digits of the analyzed ID word, now in CT(6) and CT(7), are the handler number. To reconstruct that number from the two separated digits it's necessary to multiply the second digit by ten (since it was the tens column of the original ID number) and add it to first digit. The result is a handler number from 1 to 99.

Many thanks to the man who first suggested that BASIC should include the calculated GOTO. This function, in the form ON X GOTO A,B,C, . . . Z, makes the control of program flow an easy thing. The ON . . . GOTO statement is followed by a list of BASIC line numbers; a GOTO occurs to the line in the list position specified by the variable in the statement. If the variable is the handler number, ON . . . GOTO matches that number with its location in the program and jumps to it. (Be warned, if the variable is to equal zero, no GOTO occurs and the next statement is executed. Also, if the variable exceeds the number of line numbers in the list, an error occurs.)

For a period of time, the program is under the control of one of the handlers. Depending on the function of the handler, the flow eventually returns to Executive at one of two points of entry. The first is the description subsection. After the player makes a move in the scenario, he needs to see the room into which he has moved. The descriptive portion of Executive is the logical return point. The other entry point is the command subsection. Some commands do not need a second description of the immediate environment; commands like SCORE, INVENTORY, or TAKE. After these handlers do their task, they simply return for another command.

This, then, forms the core of the adventure program. Here is a bit of review on the procedure of the program from the moment you type RUN and ENTER.

1. Initialization

- Display the game title.
- Set up variables.
- Load object status array and obstacle table.
- Create data access array.
- Move player to Room 1, clear the screen, and reset the tenacious creature.

2. Executive

●Description Subsection

  Describe the Room.
  Describe the Objects nearby.
  Describe the "tenacious" creature if nearby.
  Handle any attack from the "tenacious" creature.

●Command Subsection

  Input a command string.
  Evaluate it as one or two words.
  Look up the first word in the Word Table.
  If possible, invoke a handler from that word.

All of the preceding has simply set the stage for an effective game of Basements and Beasties. Now it's time to find out how each of the handlers actually sustain the play. The logical starting-place is to study the handlers that move the adventurer around. That is the topic of the next chapter.

Chapter 5

Traveling in the Scenario

Once the player has entered Basements and Beasties, he is placed in a room and told what it all looks like. The initiative is left with the player. What should he do? The command interpreter awaits input, and a score of handlers stand ready to do the player's bidding.

The first command an adventurer usually enters is a motion instruction. (Obviously, he wants to get a broader picture of his surroundings.) When he does, a handful of handlers come into play.

When traveling about in an adventure scenario, there are primarily three sorts of travel commands you can input. These are explicit travel commands, implicit travel commands, and magic travel commands.

Explicit travel commands give complete information on the direction of travel. As covered in Chapter 2, a player can travel in one of ten directions, eight compass points plus up and down. An explicit travel command tells the command interpreter the exact path desired. The player can type, "GO NORTH," or simply, "NORTH," or even "N." In all of these cases, the interpreter knows what is expected and can proceed to move the player along that path (assuming there are no obstacles).

Implicit travel commands, on the other hand, indicate only that motion is desired; they do not specify the direction. The interpreter must somehow perceive the direction that is intended based on the scenario. For instance, the player can be standing near a ledge. If he types, "JUMP," he has not specified a direction—but the interpreter assumes the direction is down. Similarily, if there is a room with only one door, to the north, the interpreter understands the command "EXIT" to mean the same as "GO NORTH" in this context. Implicit travel commands require more intelligence from the interpreting handler.

Magic travel commands are a stock item in adventure programs and usually come in handy in dangerous situations. These commands usually depend on a magic word or words that are immediately understood by the handler and produce a preprogrammed motion response. Magic travel is typically teleportation: rather than moving one step in a compass direction, the player is suddenly deposited in a different room, sometimes quite far from the point of origin. There are other factors involved (such as how the destination is determined), but I'll cover those in due time.

The key to these three modes of travel lies in the handlers associated with them. Therefore, let's examine these routines case by case.

EXPLICIT TRAVEL

For explicit commands of motion, there is a specific handler termed Xmove. It is the first handler in the program area designated for such routines, and Fig. 5-1 gives the code for it.

```
NAME:     XMOVE

TYPE:     HANDLER

FUNCTION:    EXPLICITLY-DEFINED MOTION


200 D=CT(8)+CT(9)*10-1:FORK=1TO10:CT(5
)=BK(K):GOSUB1000:IFD<>CT(8)ORCT(0)<>C
T(6)+CT(7)*10THENNEXTK:GOTO202:ELSEIFB
K(K)<0THEN202ELSEB=CT(9):GOTO206
202 D=D+1:GOSUB1120:IFA=22THENB=4:GOTO
204:ELSEIFA=23THENB=5:GOTO204:ELSEIFA=
0THENB=6:GOTO206:ELSECT(0)=A:CT(1)=CT(
1)+1:GOTO100
204 GOSUB1100:GOTO580
206 GOSUB1100:GOTO104
```

Fig. 5-1. Handler Xmove.

CODE			CODE		
A	B	DIRECTION	A	B	DIRECTION
0	1	NORTH	5	6	SOUTHWEST
1	2	NORTHEAST	6	7	WEST
2	3	EAST	7	8	NORTHWEST
3	4	SOUTHEAST	8	9	UP
4	5	SOUTH	9	10	DOWN

Fig. 5-2. Direction code chart. Code A is used if only one digit of storage is available for a direction.

Recall from the last chapter that the command interpreter, upon receiving an input, isolates the first word and looks it up in the word table. Upon finding the word, the ID number for that word is also retrieved. Inherent in that number is the handler number that such a command should invoke.

In the word table, there are sixteen words whose ID numbers request the attention of Xmove. These are the following:

- The eight abbreviated compass points: N, S, E, W, NE, SE, NW, and SW
- The four major compass points: NORTH, SOUTH, EAST, and WEST
- The vertical directions with abbreviations: UP, DOWN, U, and D.

Each of these sixteen words, when entered by themselves, result in the execution of Xmove, because each has an ID number ending in 01, the handler number for Xmove.

(What about the use of these words with words like "GO?" A command like "GO NORTH" is explicit because of "NORTH." But the word "GO" is handled by the implicit travel handler temporarily. You'll see this later. The explicit information is in the inclusion of the direction word "NORTH.")

The word ID number contains more than just the handler number. Digits 3 and 4 have been set aside to convey extra information, so that one general handler can respond to many individual words with varied results. In the case of these direction words, each ID number uses digits 3 and 4 to tell the handler what direction is meant. Together, those digits have a value of from 1 to 10, according to the chart in Fig. 5-2.

As in other cases of words with synonymous meanings, if a direction word is an abbreviation of another, the two have the same ID number. "SOUTH" and "S" are synonyms, and both have an ID number of 10501. The first 1 indicates that both are valid as a first word. The 05 indicates a southern course and the final 01 invokes the handler Xmove.

If you review the code for the command interpreter in the previous chapter, you'll notice that when Xmove (or any other handler) is invoked, some information is ready for use. First, the variable *N* still contains the ID number for word 1 of the input command. Second, the variables CT(6) through CT(10) still contain the five digits of *N*, isolated. Third, the strings TX$(3) and TX$(4) contain word 1 and word 2 unchanged. All of these help a given handler do its job.

THE DECISIONS OF XMOVE

Figure 5-1 shows the handler Xmove, which is probably the most overworked handler in Basements and Beasties. To aid in discussion of the code, here is a list of its tasks.

- Check the obstacle list to see if motion in that direction is in any way restricted.
- Check the travel table to see if motion in that direction is either deadly or impossible.
- Perform the motion if possible and increment the counter that keeps track of the number of steps taken.

The first task is a tough one. If the player chooses to go north, there may be an obstacle in his way. Way back in Chapter 2, you saw that there are two types of obstacles: active (like creatures) and passive (like locked doors). To keep track of these, the array BK(*n*) with special numbers that describe where the obstacles are, what directions they block, and more.

Figure 5-3 shows the way these numbers in BK(*n*) are assigned. Digits 1 and 2 of the number tell which room the obstacle is in. Digit 3 tells which direction is blockaded by the obstacle (using 0 through 9 as the ten possible directions). Digit 4 gives the message number of the line that is printed if the obstacle is encountered (message 1, 2 and 3 are set aside for obstacles).

Digit 5 indicates if there is another number in BK(*n*) that relates to this one and where it is (such as in the case of a door, which is simultaneously in two rooms). Paired obstacle numbers of this kind in BK(*n*) are always immediately adjacent to each other; digit 5 tells whether the other part of the pair is before it, after it, or simply nonexistent. (Creature obstacles occupy only one room, and thus require only one number in the BK(*n*) array.) Finally, the sign of the number indicates if the obstacle is passable or not. If the number is

SIGN	5	4	3	2	1
STATUS OF OBSTACLE + OR −	LOCATION OF MATING ENTRY 0-2	OBSTACLE TYPE 1-3	DIRECTION BLOCKED 0-9	ROOM NUMBER 1-20	

Fig. 5-3. Assignment of digits in elements of Obstacle List array BK(n).

positive, the obstacle is nonpassable; if negative, it is passable. (The door can be open or closed, for example.)

Now, Xmove knows what direction is being attempted. What it needs to do is search through every entry in BK(*n*). If it finds no entries that match the room, motion is possible. If it finds no entries that match the desired direction, motion is possible. If it finds the obstacle is passable, motion is possible, but a match in all three areas results in an obstacle.

Xmove begins by checking the obstacle list, BK(*n*), for matches with the room and direction. The direction, remember, is a part of the extra information embedded in the ID number of words like "NORTH" or "UP." The ID number is still in variable N, so Xmove needs to isolate that direction information.

The first expression in Xmove does this very thing. CT(6) through CT(10) still contain the digits 1 to 5 of the ID number, and CT(8) and CT(9) contain the direction value, from 1 to 10. The expression CT(8)+CT(9)*10 retrieves that value, but you need it in the form of 0 to 9, since that is the form used in the obstacle list. The value is lessened by one, and the result is placed in D.

Next, Xmove needs to set up a loop to test each of the numbers in the array BK(*n*). In order to compare specific digits in those numbers, each and every element needs to be broken down, using the subroutine Analyz. So, a loop must fetch a number from BK(*n*), place it in CT(5) for analysis, call Analyz, and then do the desired comparison. Xmove uses a FOR-NEXT loop with the variable *K*; there are ten entries in BK(*n*); so the loop is set to that limit.

Each time the loop selects an entry from BK(*n*), Xmove tests the entry. Is the desired direction (stored in *D*) the same as the blocked direction (stored in digit 3, or CT(8), of the obstacle number)? Also, is the present room (stored in CT(0) as always) the same as the room where the obstacle is? (The expression CT(6)+CT(7)*10 recreates the room number from digits 1 and 2.) If not, the examination loop continues with another entry from the obstacle list. If no matches are found, program control goes to line 202, which checks for other travel restrictions.

What if an obstacle match occurs? In that case there is still a final question: is it passable? Sure, there's a door here—but it may be open! The way to check is to see if the number is less than zero. If so, the obstacle is passable and can be ignored. If not, the obstacle poses difficulty and motion is prohibited.

Digit 4 of the obstacle number contains the message number to be used in printing the explanation for the difficulty. In Basements and Beasties, this digit is a 1 for creatures, a 2 for steel grates, and a 3 for doors. Message 2, for instance, says, "THE GRATE IS CLOSED AND LOCKED." Line 206 calls Mesprt to display the line and then returns to Executive.

(An obstacle can be made passable, of course, if you know how. Commands like UNLOCK for doors and KILL for creatures are discussed in their proper chapters.)

CHECKING THE TRAVEL TABLE

This is all very well and good. Perhaps there isn't any locked door in the way. Now Xmove must consult the authoritative travel table, the map of the scenario. From it, the handler can tell what room is the destination of the desired direction, or if that direction leads to some sort of horrible doom.

Figure 5-4 gives a small portion of the actual travel table, which is data block 1. In its entirety, the table has twenty lines, one for each room. For each line, there are ten numbers, plus an extra that is used by another handler for implicit travel. These ten numbers correspond to the ten possible directions. Each number is the number of the room which is the destination of a move in that direction. Thus, if a routine needs to know where the player will end up if he is in room 3 and tries to go southeast, it is simple. It finds the third line (for room 3) and the fourth number (for the fourth direction, southeast). The table says that the player will end up in room 10, provided no obstacles are in the way.

Wait a moment! What are all of those zeroes in that line? There isn't a room 0, is there? True. In addition to travel resulting in arrival at a room, travel can also result in no motion—because there may be a wall in that direction. The room number zero represents a wall, and any attempt to move in that direction results in the message, "YOU CAN'T GO THAT WAY." Plus, travel can result in death, if the player falls off a cliff or steps into a wall of flame. The unused room number 22 represents death by falling and 23 represents death by fire. If the player moves in a direction indicated by a 22 or 23 in the

```
5000 DATA1,2,2,1,1,1,1,1,0,3,9
5002 DATA2,2,2,2,2,1,1,2,0,8,9
5004 DATA0,0,4,10,0,0,0,0,1,0,8
5006 DATA0,5,0,0,11,0,3,0,0,0,4
5008 DATA0,0,0,0,0,4,0,0,0,0,5
5010 DATA0,0,0,12,0,0,0,0,0,23,3
5012 DATA0,0,0,14,0,0,0,0,0,0,3
5014 DATA0,0,0,0,14,0,0,0,2,0,8
5016 DATA9,0,16,15,9,0,0,9,0,0,7
                  •
                  •
                  •
```

Fig. 5-4. The first several lines of the travel table as it is stored in a DATA block.

travel table he dies, and a special handler called Resur (for resurrection) is called to give the player a new start.

To help Xmove search the travel table for these all-important numbers, there is a subroutine called Travec (for travel vector). Given a direction number from 1 through 10 stored in variable *D*, Travec finds the destination number in the table line for the present room and returns with that number in variable *A*. Xmove already has the direction number in *D* in the form 0 through 9; it adds one to *D* and calls Travec. (Check Chapter 3 for the discussion on Travec and how it uses Access to locate the numbers.)

Since *A* has been set to the destination number, it is easy to compare *A* to the three special case numbers, 22, 23, and 0. In the first two special cases, a death-notice message must be displayed. The variable *B* is set to message number 4 (for a fiery death) or 5 (for a falling doom), and Mesprt is called. Then the special death-handler Resur is executed from line 204. If the destination number equals zero, then the message number for "YOU CAN'T GO THAT WAY" is placed in *B* and Mesprt is called. The handler loops back to the command subsection of Executive to receive a new command.

If Xmove has managed to elude all of these special cases, then the motion finally takes place. CT(0), the room number is changed to *A*, the destination number. At the same time, the variable CT(1) is increased by one. CT(1) is the counter that records the number of steps taken, which is always of interest to players trying to get through the basement in the minimum number of steps. Xmove then terminates by looping back to the description subsection of Executive, so that the adventurer can see his new location.

See how complex mere motion can be? And that's only explicit travel.

IMPLICIT TRAVEL

Implicit travel comes into play with motion words that do not specify the intended direction of travel. For these words, there is a second handler, called Imove (for implicit move). Figure 5-5 shows the entire BASIC routine.

Imove is handler 2. In the word table there are presently five words whose ID numbers request the execution of Imove. These are IN, OUT, GO, ENTER, and EXIT.

Strangely enough, all five have identical ID numbers. The question that comes is this: how is Imove to know which direction to infer from those words? None specify extra information in their ID words.

The answer is found in the travel table. Remember that for every room there is a line; and for every line there are ten regular destination numbers, plus an unused eleventh number. That eleventh number now comes into play as the *default direction*. It is not a room number; it is a direction number from 0 through 9. In any case in which direction is not explicit, this default direction number is used.

Consider the case of room 1, the above-ground pit, which has a hole in the ground leading to the basement. Down is the default for room 1; so that IN or ENTER result in the logical motion of entering the hole. True, EXIT and OUT do not fit, and GO could be interpreted in any direction. What the default direction does is limit the amount of code necessary to handle implicit travel. Without it, for every room there would need to be a separate number for each implicit word used. There are some possible compromises, but for the moment this method of choosing direction works quite well.

Imove is understandably quite simple. There are three cases in which it is executed. In the first, the implicit-travel word can be input alone, as in GO. In the second, the implicit-travel word may be input paired with another implicit-travel word, as in GO IN. In the third, the implicit-travel word can be input paired with an explicit-travel word, as in GO NORTH. Imove can handle all three cases.

The first case is evaluated first. If the player merely types in "GO," there is a word 1 with no word 2. Thus, the string TX$(3), which holds word 2, is empty. Imove checks to see if TX$(3) is null.

```
NAME:    IMOVE

TYPE:    HANDLER

FUNCTION:    IMPLICITLY-DEFINED MOTION


220 IFTX$(3)=""THEND=11:GOSUB1120:N=A*
100+10101:GOTO108:ELSEA$=TX$(3):GOTO10
6
```

Fig. 5-5. Handler Imove.

If it is, Imove seeks out the default-direction number. Setting the variable *D* to 11 and calling Travec, the default-direction number is obtained in *A*. From this number it creates an artificial ID number, the sort of ID number that an explicit-travel word might have. The expression A*100+10101 results in an ID word that requests the explicit handler Xmove and specifies a desired direction of 1 through 10.

Finally, Imove injects this artificial ID number into Executive at line 108. At that point Executive acts as if it had received an explicit-travel command and proceeds accordingly.

If two input words are used, the second word is placed in A$. It re-enters in Executive at line 106. At that point, Executive acts as if only one word, the second one, has been input. In the case of GO NORTH, Executive now sees NORTH and has no trouble knowing what to do. In the case of GO IN Executive sees IN and eventually requests Imove again, which uses the default direction.

That takes care of two of the kinds of travel possible in the adventure program. There remains one more to consider.

MAGIC TRAVEL

Magic travel in adventure programs is usually included to help the player out of some sort of a trap or to provide a way to complete the game in the least possible steps. At root, magic travel permits the adventurer to circumvent the standard rules of scenario motion and make a sizable leap into a far distant room, ignoring any walls or obstacles that may be in the way.

This sort of travel is accomplished by the use of a magic word of some sort. Part of the challenge of an adventure game is to find out if magic travel exists and what word triggers it off. Perhaps the word is written on a wall of one of the rooms. Maybe it is in a book. Maybe one of the creatures said it at times. Whatever the case, the word is hidden somewhere and must be unearthed.

In Basements and Beasties, as you'll find, the magic word is written in a short poem on the wall of Room 6. If the player enters the command READ, he hears the poem. The magic word is AARDVARK (don't ask me why; it just sounded right). There are two ways of using it. The player can enter the command "SAY AARDVARK," and get magic travel. The player may simply type in the word "AARDVARK," and it will still work. There are limitations on the effectiveness of the word, as we'll see shortly.

Right off the bat, you can see that you need three handlers to support the use of magic travel as it has been described. You need:

- One handler to recognize the word READ
- One handler to recognize the word SAY
- One handler to recognize the word AARDVARK

The first handler to examine is the handler called READ. It starts at line 400. Figure 5-6 is the listing.

There are two cases in which the player might use the command READ: either when he is in room 6 or when he is somewhere else (how simple). In the room description for room 6, the player is informed that an oracle "HAS LEFT A MESSAGE ON THE WALL." There is no reading material anywhere else in the basement. You can expect only two responses to the command READ. If the player is in room 6, he hears the poem recited. If he is elsewhere, he hears nothing of interest. The handler READ, then, should be able to determine where the adventurer is and be prepared to print one of two messages depending on the location.

```
   NAME:    READ

   TYPE:    HANDLER

   FUNCTION:    READING OF SPECIAL

                MESSAGES

400  IFCT(0)<>6THENB=32:GOTO402:ELSEB=3
3
402 GOSUB1100:GOTO104
```

Fig. 5-6. Handler Read.

```
NAME:     SAY

TYPE:     HANDLER

FUNCTION:    SAYING OF MAGIC WORDS


460 IFLEFT$(TX$(3),5)<>"AARDV"THENB=34
:GOSUB1100:GOTO104:ELSE560
```

Fig. 5-7. Handler Say.

Line 402 of READ is a call to the subroutine Mesprt, to display the message chosen. Line 400 selects the message. If CT(0), the present room location, equals 6, the poem is printed. The poem has a message number 33. In any other room message 32 is displayed: "NOTHING HERE TO READ . . . HOW DULL!"

The poem that is displayed is no great work of art, but it does the job:

THE DANGER HERE
IS PRETTY THICK.
BUT SAY AARDVARK
YOU'LL GET OUT QUICK!

It should be noted, as an aside, that the poem is contained, as are all messages, on a one-line DATA statement. But how is it that it is displayed in four neat little stanzas like that? The secret is in how it is typed into the DATA line. The down-arrow of the TRS-80 inserts a line-feed into the text. When message 33 is being created, the programmer inserts a line-feed in between each of the four sections of the poem. The DATA statement doesn't care, but the end result is catchy when it is displayed.

The next handler to take a look at is SAY. First, if the player enters the input "SAY AARDVARK," the handler should respond exactly as if the player had simply said the magic word by itself and initiate magic travel. Second, if he enters the input "SAY XYZ," where XYZ is anything but the magic word, nothing should occur and a message should be displayed.

The second case is checked at the start of the handler SAY, in Fig. 5-7. If the player enters "SAY AARDVARK," word 2, which is kept safely in variable TX$(3), is the word "AARDVARK." The handler looks at the first five letters of the second word 2, just to see if they fit this case. The BASIC expression LEFT$(X$,n) is used to extract the desired letters of TX$(3). If a match does not occur (and woe to the player who misspells "AARDVARK"), then Mesprt is called to display message 34, which says, "NOTHING HAPPENS." Then Executive is re-entered.

Note that this sort of message is desirably noncommittal. It does not say, "I DON'T RECOGNIZE THAT WORD," even if that second word is absent from the word table. The idea is to leave the player in doubt as to whether or not that second word may still be useful. In any new game experienced adventure players try to use old magic words that they picked up from similar games. Thus, they will type, "SAY ABRACADABRA," or, "SAY OPEN SESAME," or whatever. Since this handler only states that nothing happened, it is possible (reasons the player) that the command might work in some different room or under different circumstances. This sort of ambiguity prolongs the mysteries of the game.

What if he says "SAY AARDVARK?" In that case the handler goes ahead and jumps to line 560, which is the beginning of yet another handler. This is the handler AARDVARK (see Fig. 5-8). Lines 560 and 562 actually determine whether the player experiences magic travel or not.

The limitations of magic travel vary from adventure program to program. In some games the player must be holding some particular object in order to travel. In others, he must be in a specific room. In some games, the travel amounts to a random teleportation. In others, magic travel is limited to a two-way path between two predetermined rooms.

In Basements and Beasties magic travel occurs between two rooms only: room 6 and room 1. This is helpful for two reasons. First, room 6 contains a dangerous creature who guards the only doorway out of the room. If the player wanders into the room, he finds a treasure and a trap! The only way out of the room is magic travel. Second, room 1 is the bottom of the pit room, and it is the home base of Basements and Beasties.

For any new-found treasures to be registered in the player's score, they must be smuggled out of the basement and up to room 1. It is very helpful to have a magic pathway to home base; the slower method is to travel on foot all of the way through the basement, risking an encounter with a hungry creature.

The handler AARDVARK brings about travel between these two rooms. Checking CT(0), the present room location, AARDVARK determines which way the travel should go. If the adventurer is in room 6, he is switched to room 1 simply by changing the value

```
NAME:    AARDVARK

TYPE:    HANDLER

FUNCTION:    OPERATION OF MAGIC WORD

560 IFCT(0)=6THENCT(0)=1ELSEIFCT(0)=1T
HENCT(0)=6ELSEB=34:GOSUB1100
562 GOTO100
```

Fig. 5-8. Handler Aardvark.

of CT(0). If he in room 1, he is transported to room 6. What if he is somewhere other than room 1 or 6? If that's the case, that ambiguous message 34 is displayed: "NOTHING HAPPENS." Again, the player is left with the question of under what circumstances the word "AARDVARK' works.

TRAVELING, IN REVIEW

Looking back, you have seen the three types of travel that are available to the adventurer, along with their associated handlers. These are explicit travel accomplished by the handler Xmove, implicit travel accomplished by the handler Imove, and magic travel accomplished by the handlers SAY and AARDVARK and supported by the handler READ.

Realistic travel conditions form one part of the believability of an adventure scenario. The ability to interact with objects within the scenario forms another. In the next chapter, you'll see how such interaction is effected in Basement and Beasties.

Chapter 6

Affecting the Scenario

How would you feel if you were walking around in someone's home, and you tried to pick something up, but it wouldn't budge? Just when you thought you'd gotten a good grip on that magazine, you lifted it . . . but it stayed put. Then you tried to leave; you reached out to open the front door . . . but it refused to open. How much more nightmarish could it get?

A world in which nothing can be changed is an unreal world. In order for the artificial world of the adventure program to sustain a simulated reality, the wandering adventurer must be able to bring about changes in it. Doors must open and close; objects must be movable.

Two sets of input commands are implied by this requirement of simulated reality. These are:

- Commands to unlock and open doors and to close and lock them;
- Commands to pick up and carry objects and to drop them.

For each of these commands there are associated handlers, tables, and arrays that are affected by them, specifically, the object status array and the obstacle list.

BEHIND CLOSED DOORS

In order to understand how handlers that open and close doors work, we need to review the obstacle list for a few moments. Remember that the array BK(*n*) contains a set of numbers that describe obstacles that impede the progress of the adventurer. The three types of obstacles are doors, steel grates, and creatures. Doors and grates are subject to the handlers under discussion now; creatures can only be handled by battle, as described in the next chapter.

Doors and grates are unique things, since they actually occupy two rooms at once. For this reason, each door or grate needs two entries in the array BK(*n*), one for the status of the obstacle in each room. If a door is closed and locked, it must pose an obstacle to the adventurer regardless of which side of the door he is on. Thus, whatever handler opens and closes doors and such, it must be able to change both status numbers for that door in BK(*n*).

In Basements and Beasties as in similar adventure programs, there exists a key (object 11) that unlocks doors and grates. Without this key the status of those obstacles in BK(*n*) cannot be altered. Unlike other programs, however, doors and grates exist in one of only two states: closed and locked or unlocked and open. Other programs may permit an intermediate state of "closed yet unlocked," but this seemingly simple addition complicates obstacle handling quite a bit. (That doesn't keep you from adding it if you think it's worth the trouble.)

Consider first a hypothetical handler that opens doors. Such a handler must answer the following questions:

- Did the player tell what he wanted to open?
- If he did, is that door or grate nearby?
- If it is, is the door or grate closed?
- If it is, does the player have a key?

The handler that answers these questions and opens the door is handler 5 and is called Open. Figure 6-1 provides the Open listing. There are two words in the word table whose ID numbers request the execution of Open. These are OPEN and UNLOCK. This makes sense, because to unlock a door in this program also causes it to swing open and to lock it implies that it is closed. Thus, the two words can be treated as synonymous.

Open begins by checking to see if the player provided enough information for a valid response. If the player merely types "OPEN," that may not be good enough; there may be two doors that are adjacent to a given room. Open checks for this case by looking at word 2, which is stored in TX$(3). If TSX$(3) is of null length, then Open does not bother to proceed any further. Rather, it issues the standard "play-dumb" statement, message 7, by setting the variable

```
    NAME:     OPEN
    TYPE:     HANDLER
    FUNCTION:    OPENING OF DOORS AND
                    GRATES
280 IFTX$(3)=""THENB=7:GOTO284:ELSEA$=
TX$(3):GOSUB1080:CT(5)=N:GOSUB1000:A=C
T(8):GOSUB1200:IFA=0THENB=12:GOTO284:E
LSEIFBK(A)<0THENB=13:GOTO284:ELSEIFOB(
11,1)<>21THENB=16:GOTO284:ELSEGOSUB122
0:B=12+CT(9)
284 GOSUB1100:GOTO104
```
Fig. 6-1. Handler Open.

B to 7 and calling Mesprt in line 284. Message 7 simply asks, "WHAT DID YOU SAY?" and gives the player another chance to be more lucid.

Assuming that the player did enter some sort of second word along with the key word "OPEN" or "UNLOCK," the handler tries to identify the meaning of that word 2. It calls the subroutine Idword, which begins at line 1080. Idword takes the word stored in the string variable A$ and searches the word table for it. If it is found, it returns with the ID number for that word in the variable *N*. If it is not in the program's vocabulary, it returns with *N* set to zero. Open saves word 2 in A$ and lets Idword loose on it.

When Idword is finished, Open is interested in the individual digits of the ID number stored in *N*. Since this is so, it calls Analyz to break *N* up into digits. Analyz takes the contents of CT(5) and places digits 1 to 5 in CT(6) to CT(10). Open sets CT(5) equal to N, and Analyz does the rest. Note, for the moment, that if *N* equals zero (because the second word was not found in the word table, Analyz simple places zeros in all of the variables CT(6) to CT(10).

Now that Open has all of the digits laid bare, it is interested in only one of them: digit 3. Recall that for objects the digits 1 and 2 of the object's ID number represent the object number. If, for example, you look up the word "SPIDER" in the word table, the ID number has the value 15 in digits 1 and 2, because the spider is object 15 in the list of objects for Basements and Beasties.

Things like doors and grates, however, are special. They are not objects in the regular sense; they cannot be carried away or dropped. Thus, there is no object number for a door or a grate.

Rather, they are assigned a special object number of 17. Digits 1 and 2 of the ID numbers for the words "DOOR" and "GRATE" have a value of 17. Later, you'll see that when the player tries to pick up and carry anything with an object number of 17, the program refuses, telling him that "IT IS IMMOVABLE." This prevents some pretty embarrassing program inconsistencies!

If the ID number of word 2 that Open just analyzed does not have a useful object number, what good is the ID number at all? The other digits do not have any designation, do they? The answer is, yes, they do. There are three types of obstacle, remember. It can be very useful to Open if the ID number can convey which type of obstacle. Only for the two words "DOOR" and "GRATE," digit 3 of their ID number is assigned to be the obstacle type: type 2 if it is a grate and type 3 if it is a door. (Type 1 is a creature, but you don't open and close creatures.)

The reason that Open is interested in the obstacle type is simply that the obstacle-type number is used in the entries of the obstacle list, BK(*n*). There are two questions that Open needs to answer from the obstacle list: (1) is there any obstacle in this room and (2) if so, is it the same obstacle that the player wants to open or unlock?

Each entry in the obstacle list contains the answer to both of these questions. Digits 1 and 2 of each entry give the room number where the obstacle is, and digit 4 is the type number, 1, 2, or 3. So the handler Open now must search the obstacle list and do two comparisons. First, it must find any entries that match the present room number, which (as always) is in CT(0). Second, of those entries it must find any entries whose obstacle type matches the type number presently in CT(8), the obstacle input by the player as the second word of his OPEN command.

There exists a handy subroutine to search the obstacle list. It is called Ckobs (as in check obstacles), and it is given in Fig. 6-2. Essentially, it takes each and every entry in BK(*n*), breaks it up into its digits, and performs these two comparisons. If it finds such an entry, it returns with the position of the entry placed in the variable A. If no matching entry is found, *A* is set to zero. Using this value *A*, Open can find and change the appropriate entries in the obstacle list, BK(*n*).

Ckobs begins by setting up a FOR-NEXT loop of from 1 to 10, since there are ten entries in BK(*n*). Each entry is broken down by a call to the subroutine Analyz (using GOSUB 1000). The handler Open has previously set the variable *A* to the obstacle type for which it is looking. So Ckobs compares the room number and obstacle type

```
NAME:    CKOBS

TYPE:    SUBROUTINE

INPUT:   A = TYPE OF OBSTACLE (1 - 3)

OUTPUT: A = OBSTACLE LIST ENTRY

         NUMBER IF FOUND

         A = 0 OTHERWISE
1200 FORQ=1TO10:CT(5)=BK(Q):GOSUB1000:
IFCT(6)+CT(7)*10<>CT(0)ORCT(9)<>ATHENN
EXTQ:A=0:ELSEA=Q
1202 RETURN
```

Fig. 6-2. Subroutine Ckobs.

with every element of BK(*n*). The expression CTT(6)+CT(7)*10 recreates a room number from digits 1 and 2 of the obstacle list entry. If this value doesn't match the present room number in CT(0), or if digit 4 in CT(9) doesn't match the obstacle type stored in the variable *A*, the FOR-NEXT loop continues the search. If the loop runs out without finding a match, *A* is set to 0 and the subroutine returns. If a match is found, then *A* is set equal to *Q*, the variable used for the FOR-NEXT loop. Thus, if the fourth entry is a match, *A* equals 4.

Now to answer a question you may be keeping. Awhile back, a word-table search was made to find word 2. If a word is not found in the word table, the handler Open cannot check the word. If a player types in something like "OPEN CUCUMBER," what is to keep the handler from making an erroneous response?

Ckobs filters this out. Remember that if a word is not found in the word table, the subroutine Idword returns a zero. This breaks down into five zero digits. When Open calls upon Ckobs to perform the two comparisons of room and obstacle type, a match cannot occur. Why? Because an unrecognized word 2 would be requesting to open an obstacle of type zero! Such an obstacle doesn't exist; no entry in BK(*n*) has an obstacle type of zero. So the response to a command like "OPEN KANGAROO" is the same as to a command like "OPEN DOOR" in a room with no doors.

The handler Open now has an obstacle list entry number from 1 to 10, or zero if no entry is found that matches the command request.

Open begins to act on this new information. What if no such obstacle exists? If so, *A* is a zero. Open tests for this and calls Mesprt to display message 12, which reads, "I SEE NOTHING OF THE SORT HERE!"

Next, Open must decide if the door or grate needs to be opened. Obviously, if it is already swinging in the breeze, it is ridiculous for the handler to go through the act of opening it all over again. The way that Open determines this case is by referring to the obstacle list entry. *A* equals the position of the entry that was found, and BK(*A*) is the entry itself. The obstacle list indicates whether or not an obstacle is passable using the sign of the entry. That is, if the entry is a negative number, then the obstacle is passable: the door or grate is unlocked and open. Otherwise, it is closed and locked and needs to be opened. Open checks to see if BK(*A*) is less than zero, and if it is, it calls for the display of message 13, which reads, "YOU DON'T NEED TO."

The final contingency is the possession of the key. Without the key, which is object 11, no door or grate can be opened. The key must be in the player's possession; that is, he must be carrying it. It cannot simply be lying nearby in the room. Open checks the object status array to find where the key is. The variable OB(11,1) betrays the key's room number at that time. Anything that the adventurer is carrying is assigned a room number of 21. Thus, the player can only open the door or grate if OB(11,1) equals 21. If it does not, the handler calls for the display of message 16, which reads, "YOU HAVE NO KEY!"

Once Open manages to execute all these steps, it is ready to unlock and open the door or grate. To do this, Open calls upon a subroutine called Revobs (for reverse obstacle), which is given in Fig. 6-3. Given an obstacle-list entry number from 1 to 10 in variable *A* Revobs performs two functions: it reverses the sign of the entry indicated by *A*, and if a corresponding entry exists in the list, it reverses the sign of that entry as well.

Note that Revobs reverses the sign of the entry. That means that if the door or grate is closed, it will be opened. Revobs can also close open doors. Revobs comes in handy to input commands like LOCK GRATE. Note also that it finds a corresponding entry (if there is one) and complements it. That way, a door becomes open on both sides, in both rooms it connects. If there is no corresponding entry (as in the case of creature obstacles), Revobs performs only the first function. REVOBS also is used to make unpassable creatures passable, when we discuss battle commands in the next chapter.

```
   NAME:    REVOBS

   TYPE:    SUBROUTINE

   INPUT:   A = OBSTACLE LIST ENTRY

            NUMBER

   OUTPUT:  THE STATUS OF THAT ENTRY AND

            OF ITS MATING ENTRY (IF ANY)

            ARE COMPLEMENTED
1220 BK(A)=-BK(A):CT(5)=BK(A):GOSUB100
0:IFCT(10)=1RETURNELSEBK(A-1+CT(10))=-
BK(A-1+CT(10)):RETURN
```

Fig. 6-3. Subroutine Revobs.

The first function is simple. Variable *A* already carries the obstacle list entry number. So Revobs negates the variable BK(*A*). The second function takes some figuring. In the obstacle-list entry digit 5 is assigned the task of telling routines whether or not there is a second entry, and if so, where it is. Remember that the two paired entries for a door or grate in the obstacle list are always immediately adjacent one to the other. Digit 5 allows three possibilities, indicated as follows by a number from 0 to 2:

0. There is a corresponding entry immediately before this one.
1. There is no corresponding entry; this is the only one.
2. There is a corresponding entry immediately after this one.

The numbers 0, 1, and 2 were not chosen arbitrarily. Revobs already knows that entry BK(*A*) needs to be changed. Now, the entry BK(A–1) or BK(A+1) or neither needs to be changed. Now, the previous element can be expressed as (A–1)+2. Thus, Revobs can use the numbers 0, 1, and 2 to identify the entry number of the corresponding entry, if one exists.

Study Listing 6-3 to see how this is done. CT(10) contains digit 5: the numbers 0, 1, or 2. REVOBS returns if this is a 1, because it has already complemented the sign of entry CT(A). Otherwise, Revobs complements entry BK(A–1+CT(10)), which is the corresponding entry either before or after it. Then the subroutine is ended and returns to Open, which called it.

Someone is bound to ask this question, so I'll answer it now. Since there are only ten entries in the obstacle list, why not use digit 5 as a number from 0 to 9, corresponding to the ten entries? Then, an entry could specify exactly where its mate is, and the corresponding entry would not need to be right next to the first one.

The answer is that this automatically limits the size of the obstacle list to ten entries, with no room for expansion. The present Basements and Beasties has only two doors, one grate, and four creatures as obstacles. That's really a bit skimpy. The present system using relative location of paired entries allows the obstacle list to be as large as need be.

The handler Open has one last task after Revobs is finished, and that is to inform the adventurer that the door or grate has been opened. Digit 4 of the obstacle-list entry is a number from 1 to 3, indicating which kind of obstacle has been changed. In the message block in memory, the messages announcing the opening of a door or grate are placed next to each other, in just the right order to simplify matters. Obstacle type 2 is a grate, and type 3 is a door; so the message for the opening of a grate precedes the one for a door. The expression 12+CT(9) results in a value of 14 for a grate and 15 for a door. Message 14 states, "WITH A CREAK, THE GRATE FALLS OPEN." Message 15 says, "THE DOOR SWINGS OPEN WIDE." Notice that in both cases, the message is the same whether the original command was "OPEN DOOR" or just "UNLOCK DOOR." Either command has the same result.

LOCK THE DOOR BEHIND YOU

The other handler that relates to doors and grates is called Close. It is handler 6, given in Fig. 6-4. In the word list the two words whose ID numbers request the execution of Close are "CLOSE" and "LOCK."

In many ways, Close operates exactly like Open, with a few simplifications. The questions that Close must answer are:

- Did the player tell what he wanted to close?
- If he did, is that door or grate nearby?
- If it is, is the door or grate open?

If you are sharp-eyed, you noticed the one important difference between Open and Close (other than the end result). That is the requirement of a key. To close and lock a door or grate, the adventurer does not need the key. It simply swings shut and, as the accompanying message reads, "THE LOCK CATCHES."

```
NAME:     CLOSE
TYPE:     HANDLER
FUNCTION:    CLOSING OF DOORS AND
                 GRATES
300 IFTX$(3)=""THENB=7:GOTO304:ELSEA$=
TX$(3):GOSUB1080:CT(5)=N:GOSUB1000:A=C
T(8):GOSUB1200:IFA=0THENB=12:GOTO304:E
LSEIFBK(A)>0THENB=13:GOTO304:ELSEGOSUB
1220:B=17
304 GOSUB1100:GOTO104
```

Fig. 6-4. Handler Close.

Close performs the first decision by checking word 2. If word 2 in TX$(3) is nonexistent, the message "WHAT DID YOU SAY?" is displayed. Otherwise, Close takes the word in TX$(3) and passes it to Idword in the string variable A$. Idword returns with the word ID number stored in the variable *N*. Close calls Analyz to isolate the five digits of the ID number. Then it takes digit 3, the obstacle type, and lets the subroutine Ckobs determine if the obstacle intended by word 2 is really there in the room or not. If not (as indicated by a value of zero in variable *A*), message 12 is displayed: "I SEE NOTHING OF THE SORT HERE." Finally, the sign of the entry is checked. If it is positive, then the door or grate is already closed and locked, and message 13 tells the player, "YOU DON'T NEED TO."

If the input command stands valid after all three tests, Close goes ahead and reverses the status of the obstacle using the subroutine Revobs. The opened door or grate is set to a closed condition by the changing sign of the obstacle list entry, along with a change of the corresponding entry in the list.

When the time comes to tell the player what has been done, Close does not make a distinction between doors and grates, as Open did. Rather, the general message 17 is used, which reads, "IT SLAMS SHUT AND THE LOCK CATCHES."

One intriguing final note should be made about the difference between the handlers Open and Close. Open requires a key, and Close does not, as we have seen. This means it is quite possible for a poor, misguided adventurer to walk through an open door into a room with only the one exit, and slam the door shut behind him, all without a key. Both rooms 6 and 11 are traps like this, if a player is so foolish. Room 11 does provide an out, though; the magic word

"AARDVARK" teleports the player to freedom. Room 6 can be a terrible place to spend the remainder of one's game!

TAKE THE TREASURE AND RUN

Now that I've covered the specific scenario interaction affecting doors, I can move on to the general set of commands controlling carrying objects. The simple actions of picking up and dropping articles are not so simple after all. What objects are movable? How much can the adventurer carry? Questions like these must be answered by the relevant handlers.

Two handlers relate to the tasks of object-toting. These are Take and Drop, and they are invoked by the corresponding command words, TAKE and DROP, followed by the name of the object. Two other words, STEAL and THROW are synonyms with the first two command words, respectively.

Let's look at Take first. Logically, a handler to bring about the picking-up of objects must answer the following list of questions and act accordingly:

- Does the adventurer already have too much to carry?
- Does the adventurer command ungrammatically?
- Does the adventurer want to take a creature?
- Does the adventurer want to take something immovable?
- Does the adventurer already have the object in his sack?
- Is the object requested either nonexistent or not in that room?

The first question has to do with the maximum amount an adventurer can carry. In Basements and Beasties this maximum is set strictly on the basis of quantity. An adventurer can only carry five objects, regardless of size or shape. This is unrealistic in some ways, but it is simpler to handle.

If the adventurer could carry more than five objects, each and every movable object would have to be assigned a mass number or something of that sort. Then, the handler Take would determine its response by adding up all of the mass numbers of the objects now carried and comparing the result to some arbitrary maximum. If you care to do this, it should be a simple matter to assign the unused elements of the object status array, OB(X,0), as object mass numbers ranging in value from 0 to 255. Then a maximum total mass of around 500 could be set to limit what the adventurer carries. Small objects like the coin and the key would have mass numbers in the 50s, and heavy objects like the golden cube would have a value of over 100. An example of this method is provided for you in Chapter 10.

You might ask what is the purpose of a carry-limit anyway. The primary reason is to require the adventurer to make several successive trips into the basement in order to get all treasures out. If he could carry anything and everything, he would make one long excursion, get everything, get back to home base, and end the game. With a maximum limit based either on quantity or mass, he must forever fight his way back to the entrances—that adds to the challenge.

At any rate, the present version of Basements and Beasties sets an upper limit of five objects. The variable CT(2) is set aside to keep track of the number of articles the adventurer has. The handler Take must check to see if CT(2) is already at its maximum of five.

Figure 6-5 gives Take. The first question is answered by comparing the value in CT(2) with the value of five. If CT(2) equals or surpasses the maximum, Take refuses to pick up the requested Object. It notifies the player of this refusal by setting variable *B* to 36 and calling the subroutine Mesprt. This prints message 36: "YOUR ARMS ARE FULL . . . YOU CAN CARRY NO MORE." If CT(0) is less than five, though, Take proceeds to consider the other questions.

The next question has to do with the player's grammar. The command has taken the form, "TAKE X," where X is some word. In order for the handler to know which object to pick up, it must try to define that word *X*. It must submit that word to a search of the word table to find it in the vocabulary.

The grammar problem is this: what if the second word in the command is in the word table but is not an object? For instance, a player might type "TAKE OPEN." The word "OPEN" is in the word table—but it is a verb, not an object. The handler should not permit such an ungrammatical possibility.

Fortunately, the program can determine between valid objects and verbs. Each word in the word table is, of course, paired with its ID number. This ID number has a one in digit 5 if the associated word is a verb. That is, any word with an ID number of 10,000 or greater is a verb. Thus, the handler Take finds the ID number of the second word of the command and checks it against a value of 10,000.

Take uses the subroutine Idword to obtain the ID number. The second word of the command is stored in TX$(3). By setting A$ equal to TX$(3) and calling Idword, the variable *N* is set to the value of the ID number. If the word is not found in the word table, *N* equals zero.

The handler compares *N* to 9999. If *N* is greater, the player entered an ungrammatical command. The result is the display of

```
NAME:    TAKE

TYPE:    HANDLER

FUNCTION:    PICKING UP OF OBJECTS

240 IFCT(2)>=5THENB=36:GOSUB1100:GOTO1
04:ELSEA$=TX$(3):GOSUB1080:IFN>9999THE
NB=7:GOTO242:ELSEIFN>12ANDN<17ORN=18TH
ENB=40:GOTO242
241 IFN=17THENB=8:GOTO242:ELSEIFOB(N,1
)=21THENB=9:GOTO242:ELSEIFOB(N,1)<>CT(
0)ORN=0THENB=12:GOTO242:ELSEOB(N,1)=21
:B=11:CT(2)=CT(2)+1
242 GOSUB1100:GOTO104
```

Fig. 6-5. Handler Take.

message 7, which asks, "WHAT DID YOU SAY?" If *N* is less than 10,000, the command is at least grammatical, though it remains to be determined whether or not the command can be executed.

The third question is asked because of wise-guy adventurers. Almost certainly, someone will try to pick up and carry a creature. Before I added this consideration, I had a play-tester who could not get past the giant mantis. So what did he do? He carried the silly creature out of the room! After groaning longly and loudly, I interpolated this third question.

There are, of course, two kinds of creatures: the passive guard creatures and the more dangerous tenacious creature (the Orc). Passive creatures have object numbers from 13 to 16. The Orc, although his position information is kept in OB(0,1), has an object number of 18 in the ID number of the word table. When the handler Take finds the ID number for the object to be carried, it must compare that number to those of the creatures.

If *N*, the ID number, is both greater than 12 and less than 17, then a passive creature is intended. Or, if *N* equals 18, the Orc is intended. In either case, the command is rejected by a call to Mesprt for message 40: "YOU MANIFEST SOME PRETTY SUICIDAL TENDENCIES, FELLA!" That'll keep them from dragging your dragons away!

The fourth question relates to immovable objects. Every room has a somewhat elaborate description, telling its features, its colors, and so on. In some cases such a description may mention the presence of some article which nevertheless is not an object. Note, for instance, that the description for room 14 includes the statement, "THERE ARE COBWEBS EVERYWHERE." Now, suppose that the player typed in the command, "TAKE COBWEBS." Since Cobwebs are not an object with an element in the object status array, how can such a command be executed? If the word COBWEBS is left out of the word table altogether, the response to that command would be "I SEE NOTHING OF THE SORT HERE," which would sound ridiculous, since the room description just said that they were there. Yet, if you put the word COBWEBS into the word table, what ID number do you give it? You can presumably expand the object status array to cover all these descriptive articles, but that would be wasteful.

To simplify the situation, all descriptive articles are added into the word table. Rather than unique object numbers, however, all are assigned the value 17 in their ID number. The adventure program knows how to treat all objects 17—as recognizable, but less than true objects.

The handler Take checks to see if the article within reach is an object 17. If it is, the command is rejected. Unfortunately, there is little logical ground for refusing the command. If there are cobwebs there, why can't the adventurer take them? So, rather than giving any real explanation, Mesprt is called to display message 8, which avoids the subject but remains firm: "YOU TRY UNSUCCESSFULLY . . . IMMOVABLE!" Granted, this is less than satisfying, but the only simpler choice is to write room descriptions that don't even hint at furniture or articles other than legitimate objects. That can result in a boring scenario.

The next question checks to see if the command is even necessary. Maybe he already has the object and doesn't need to take it! How can you tell? All objects that are in the player's possession are given a location of value 21. That is, they no longer reside in the room where he stands; they reside in room 21, which is the player's sack. If the adventurer already has the object, the handler Take knows by checking the object status array.

Since the variable *N* gives the object number of the article, the element OB(N,1) gives the object's physical location. If OB(N,1) is equal to 21, the command is rejected and message 9 is displayed: "YOU ALREADY HAVE IT!"

The final question is whether or not the requested object is available for the taking. There are two cases to handle. In one case, the object may be in another room altogether. In the other, the requested article may not exist in the word table. In either situation, the handling is the same—the handler replies that it doesn't see the article nearby.

As in the previous question, since *N* equals the object number, OB(N,1) gives its location. CT(0) tells the room number where the adventurer is. Thus, if OB(N,1) doesn't equal CT(0), the object simply is not there. On the other hand, if the player asked to pick up an article unknown to the program's vocabulary (as in "TAKE WOMBAT"), the variable *N* would equal zero, because that is the result of an unsuccessful word table search using the subroutine Idword. If *N* equals zero, or if the other case occurs, the command is rejected with message 12, which says "I SEE NOTHING OF THE SORT HERE." Note that this does not reveal the program's ignorance of the article mentioned in the command; the player *may* find a Wombat elsewhere!

If the handler Take gets through all six of the above contingencies it is ready to perform its function. It does this in three steps. First, the object must be transferred into the player's possession. This is done by removing it from the room and placing it in the carry-sack. The variable OB(*N*,1) is set equal to 21 to effect this transfer. Second, the program must keep track of how many articles the adventurer is now carrying. Take performs an update by adding one to the present value of CT(2), which records his inventory total. Finally, the player must be notified of the success of the transaction. For this, message 11 is printed: "OKAY." As usual, a deceptively simple message is used, obscuring the complex decision-making that led up to it!

That takes care of picking up objects. Now we need to examine how objects are dropped back into the room.

DROP THAT TREASURE!

There are two key words in the word table that are treated as synonymous and relate to the dropping of carried objects: DROP and THROW. Both of these invoke the handler Drop, which is given in Fig. 6-6.

The operation of Drop is similar to, but simpler than, that of Take. There are three questions this handler seeks to answer before it can execute the command:

- Does the adventurer command ungrammatically?
- Does the adventurer have the object in his carry-sack?
- Is the adventurer dropping the Enchanted Grenade?

All three questions depend on the object number of the article to be dropped. Idword is therefore called to locate the word stored in

```
  NAME:    DROP
  TYPE:    HANDLER
  FUNCTION:   DROPPING OF OBJECTS
260 A$=TX$(3):GOSUB1080:IFN>9999THENB=
7:GOSUB262:ELSEIFOB(N,1)<>21THENB=10:G
OTO262:ELSEIFN=12THEN540:ELSEOB(N,1)=C
T(0):B=11:CT(2)=CT(2)-1
262 GOSUB1100:GOTO104
```

Fig. 6-6. Handler Drop.

TX$(3) somewhere in the word table and to place the word's ID number in variable *N*. (For objects, the ID number equals its object number.)

The first question is handled just as in the handler Take. If the player has used a verb as the object of the command DROP, the value of *N* exceeds 9999; that is, it is 10,000 or greater, since digit 5 is set to one for verbs. If this happens, message 7 gives the player another chance with "WHAT DID YOU SAY?"

The next question is handled analogously to the handler Take, with converse results. In this case the command is rejected if the player does not have the object in his possession. If OB(N,1) does not equal 21, the object is not in the carry-sack. Mesprt is called tc print message 10: "YOU DON'T HAVE IT!"

The final question cannot be fully explained until the next chapter; there is one object that responds very strangely to the action of dropping or throwing, and that is the Enchanted Grenade. It's object number is 12; if the handler Drop finds that object 12 is being thrown, it refers the whole affair to line 540, which is the start of the handler called Bomb. You'll see later that a number of things may happen when Bomb is invoked, but that is another story.

With these exigencies considered, the transfer can now occur. As before, there are three steps. The object's location is changed, by setting OB(N,1) equal to the room number stored in CT(0). The inventory total in CT(2) must be updated by subtracting one. Lastly, the simple message "OKAY," message 11, is displayed.

The adventurer is making gradual progress. A few chapters back he could merely walk about and look at things. Now he can touch those things, take them with him, and open and close doors. In the next chapter, the adventurer learns to defend himself against the creatures that roam unchained in the dark corridors of the basement.

Chapter 7

Battling the Enemy

The danger factor differentiates an adventure program from a mere Easter egg hunt. If all the adventurer has to do is wander around and find treasures, there is no challenge! There must be something to defy his attempts, something to hinder his progress, even to threaten his life. That is why adventure programs have creatures.

Various programs handle their creatures differently. Some creatures wander aimlessly about the scenario, bumping into the adventurer at random. Some have a stationary post that they guard continually. Some do not attack unless threatened. Others cannot be slain by normal weapons. Battles may be decided on purely random factors, or a record may be kept of the combatant's strength levels to determine who should rightly be the victor.

Basements and Beasties has a combination of many of these variations in its method of battle simulation. An attempt is made to keep the algorithms simple while maintaining the illusion of an actual struggle. There are three classes of battle in the program:

- Attack/retaliation with certain passive creatures,
- Special weapon against certain other passive creatures
- Defense/offense against the tenacious creature

Figure 7-1 shows the beasts that wait in the wings. You recall that there are really two basic kinds of creatures in any adventure program. One type might be called passive creatures. Their main purpose is to guard or block some passageway in the scenario. As such, they are also bona fide obstacles and are present in the

TYPE	CREATURE	WEAPON	ROOM
PASSIVE	GIANT MANTIS	AXE	4
PASSIVE	HUGE IGUANA	AXE	18
PASSIVE	WHITE SPIDER	GRENADE	14
PASSIVE	NAMELESS TERROR	GRENADE	6
ACTIVE	ANGRY ORC	AXE	

Fig. 7-1. The creatures of Basements and Beasties.

obstacle table. Other obstacles, like doors, are rendered passable by the "OPEN" command. Passive creatures are rendered passable by battle. They do not attack on their own, but if they are attacked, they always retaliate. Since they are not immediately hostile, it is not necessary for the adventurer to engage them in battle. However, the player gains points for every creature killed, and there are certain treasures he can never retrieve without passing a passive creature.

To add to the challenge, not all passive creatures can be beaten in the same manner. Of the four passive creatures, there are two subsets of two each. One set may be engaged in the attack/retaliation cycle and eventually slain. The other set is totally immune to the standard weapon (the Axe), but may retaliate nevertheless. The only way to kill these two creatures is with the Enchanted Grenade.

Separate from the passive creatures is the much more dangerous tenacious creature, the Orc. Class 3 battle is called defense/offense because the Orc attacks without provocation. He can be killed in the usual manner (with the Axe), but he follows the adventurer from room to room. The player's only escape is either to slay the Orc or run back above ground, where the Orc can not follow.

A number of sections of code interact to support this battle simulation. I already discussed one of these in the examination of Executive a few chapters ago. That section handles the motion of the tenacious creature, whether he attacks, and how successful he is. The standard battle handler is called Fight, and it controls the outcome of any encounter with a creature that can be killed by the Axe. Another handler, called Bomb controls the effect of the Enchanted Grenade on those two creatures that have tough skin and are Axe-resistant. Finally, there is a handler called Resur, which is invoked when the adventurer dies. It resurrects him outside of the basement, adjusts the score, and handles a few other details.

FIGHT THE GOOD FIGHT

In the word table there are three words that relate to standard battle. These are KILL, FIGHT, and SLAY. All three are treated synonymously and invoke the same routine: handler 7, the section called Fight. Figure 7-2 gives the code for it.

The handler Fight answers the following questions when it is invoked:

- Is the tenacious creature there?
- Is any passive creature there?
- Is the standard weapon at hand?
- Is the creature resistant to the standard weapon?

Assuming that a battle does ensue, Fight moderates the skirmish according to the following set probabilities:

- There is a 70 percent chance that the creature is killed this turn.
- If this is not a tenacious creature, there is a 30 percent chance that the adventurer is killed in retaliation.

Note that the 30 percent retaliation figure applies only to passive creatures. The handler Fight does not cause the tenacious Orc to fight back. Rather, Executive handles the Orc's response.

The first question to be asked is whether or not the enemy being challenged is in fact the Orc. Recalling the discussion of Executive, the unused elements of the object status array, OB(0,0) and OB(0,1), are set aside for the tenacious creature. OB(0,1) gives his location, and Executive moves him around randomly. As long as the Orc is not in the same room as the player, OB(0,0) is kept at a value of zero. If the Orc stumbles across the adventurer, however, OB(0,0) is set to one. From then until the player either kills the Orc or escapes to the surface, the Orc tracks the player from room to room.

The element OB(0,0), then, is an easy way to tell if the Orc is around. If it equals one, Fight automatically assumes that the player is trying to slay the Orc. The handler proceeds down to the next line to handle the other questions.

This raises an intriguing consideration. If the player enters the command KILL, and if there are two creatures in the room, Fight always defaults to the tenacious creature Orc first. In this way, the player need not specify the creature's name in the heat of battle, and

```
NAME:     FIGHT

TYPE:     HANDLER

FUNCTION:    BATTLE WITH CREATURES


320 IFOB(0,0)=1THEN322ELSEFORK=13TO16:
IFOB(K,1)<>CT(0)THENNEXTK:B=41:GOSUB11
00:GOTO104
322 IFOB(10,1)<>21THENB=23:GOTO326:ELS
EIFK=15ORK=16THENB=24:GOTO324:ELSEX=RN
D(100):IFOB(0,0)=1THEN328ELSEIFX>70THE
NB=26:GOTO324:ELSEOB(K,1)=0:A=1:GOSUB1
200:GOSUB1220:B=25:GOTO326
324 GOSUB1100:B=27:GOSUB1100:X=RND(100
):IFX<40THENB=29:GOSUB1100:GOTO580:ELS
EB=28
326 GOSUB1100:GOTO105
328 IFX>70THENB=26:GOSUB1100:GOTO112:E
LSEOB(0,0)=0:OB(0,1)=0:B=25:CT(4)=CT(4
)+25:GOTO326
```

Fig. 7-2. Handler Fight.

the handler is not confused by two types of creatures at once. This assumption is not too hard to accept, since the Orc is hard to ignore, and only a fool would waste his time provoking a sleepy, passive creature while the Orc is leaping at his throat all the time.

The next question is whether any other creatures are present in the room. To check this out, a FOR-NEXT loop is set up to scan the object status array. Passive creatures are objects with numbers in the range from 13 to 16. For each of the four creatures, it's location is compared to CT(0), the location of the player. The loop continues until a match is found. If the match occurs, program control drops to the next line for further questions. If no match occurs, i.e., if no passive creatures are there, the loop runs out. The handler can only assume that the poor delirious adventurer has tried to attack and kill a rock or something. It sets the variable *B* to 41 and uses the subroutine Mesprt to display the message, "SAVE YOUR STAMINA, TURKEY! I SEE NO REAL THREAT!"

I might add that this section of Fight was added to cover up an embarrassing situation. I had a play-tester who sat down and found this flaw. He entered a room that had no creatures at all, entered the command Fight, and suddenly, a nonexistent creature appeared, leapt at his throat, and killed him! Programmer, beware. If you haven't thought out all of the possible options, your player will stumble on a few beauties!

The third question is whether the player even has a weapon with which to fight! The standard weapon is object 10, the Axe. If it is in the adventurer's carry-sack, OB(10,1) equals 21, the location number indicating possession. If not, the player is unarmed, and the handler responds with message 23, which asks, "WITH WHAT WEAPON?" Notice that this question procludes use of the Enchanted Grenade through this handler. If the player wishes to bomb his opponent, he must enter the specific command key word BOMB.

Even if the player has the Axe, there are two creatures with skin too tough to harm. These are the White Spider and the Nameless Terror, which have object numbers of 15 and 16. If the adventurer swings his Axe at either creature, the handler displays message 24, "YOUR AXE SWINGS ARE DYNAMIC . . . BUT INEFFECTIVE!" The creature suffers no harm, but the handler continues on from that point to the next line, which controls retaliation. Thus, a player may die in learning the secret that the Axe cannot kill these two beasts.

With all four preliminaries out of the way, the player's attack can be simulated. The variable *X* is randomly set to some number from 0 to 100. This number provides the probability percentage for the success or failure of his attack. Before that probability can be evaluated, however, the program forks in two possible directions. If the creature is tenacious, his doom or survival is handled differently than that of a passive creature. Line 328 takes care of this; and you'll see it in a moment.

For the passive enemy, though, the random percentage is examined. If *X* is greater than 70 (a 30 percent chance), then the attack was unsuccessful and the creature survived. If this is the case, message 26 is displayed: "MISSED IT! FIE!" Program control proceeds to the next line, as the creature is given a chance to retaliate.

What if *X* is less than or equal to 70? If so, the Axe has met its mark, and the handler must remove the creature. This requires a few steps. First, the creature must be removed from the room so that it is not described by the description subsection of Executive. The most effective way to do this is to move it to room 0, the "nonexistent" room. OB(K,1) gives the location of the creature, since if this is a passive creature, K equals its object number due to the FOR-NEXT loop up in line 320. Setting OB(K,1) to zero sort of dispatches the creature to limbo.

That's not all. The passive creature is not just an object. It is also an obstacle, with an entry in the obstacle list. The handler Fight must change this obstacle list entry so that the player can move freely through the passageway previously guarded by the creature. This is done through two subroutines. The first, Ckobs, finds the obstacle list entry of the creature. The second, Revobs, toggles the status of the entry from unpassable to passable. (A more detailed description is available in the previous chapter.)

The subroutine Ckobs, located at line 1200, needs to know the obstacle type and the present room number in order to find the entry. The room number is always in CT(0); the obstacle type must be stored in variable *A*. The handler sets *A* equal to 1 (the obstacle type for creatures) and calls Ckobs. When the subroutine is done, *A* is set to the entry number, which is a number from 1 to 10.

The subroutine Revobs, located at 1220, needs to know the obstacle list entry number in order to change the entry's status. It expects this number to be in variable *A*. Fortunately, Ckobs used *A* for that number, so no preparation is necessary; Revobs can be called right after Ckobs. When that subroutine is done, the obstacle list entry indicates that the passageway is open for travel.

The final step in handling the slaying of a passive creature is the death notice. The subroutine Mesprt displays message 25, which reads, "YOUR MAGIC AXE CONNECTS! THE CREATURE VANISHES IN A PUFF OF FOUL SMOKE!" The handler is then finished and returns to the command subsection of Executive.

The past few paragraphs have dealt with attacking a passive creature. Before looking at its retaliation, let's see what happens if the enemy is the tenacious creature. Line 328 handles the attack in this case. The variable *X* is still some random number from 0 to 100 as before. If *X* is greater than 70 (a 30 percent chance), then the Orc has avoided the player's Axe. Message 26 is shown ("MISSED IT! FIE!"), but instead of skipping to a routine to provide retaliation, program control leaps back to the Executive, right before the description of the Orc. That passage of Executive causes the Orc to launch his own offensive. Splitting things up this way provides a means for Orc to attack repeatedly, relentlessly, possibly every turn, making him the toughest of creatures to beat.

If the probability value in *X* is less than or equal to 70, the Orc has met his match. Note, though, that his demise is handled uniquely. The variable OB(0,1) controls where he is; this is set to zero, sending him to the "non-room." The variable OB(0,0) controls his actions. This is set to zero, which puts him into a waiting mode. In short, the Orc never really dies. He is just temporarily sent to room 0. Executive sets him traveling again on a random basis. Thus, the adventurer soon meets the Orc again in his travels. The best way to think of this feature in the scenario is that the basement is full of wandering Orcs, and that each one that comes along is a new one. This randomly reoccurring danger adds to the interest of the game.

Even though the Orc (or an Orc, if you will) returns, the message printed is a death notice identical to that for vanquished passive creatures: message 25.

After each Orc is killed, the variable CT(4) is increased by a factor of 25. The next chapter explains this, but for now, recognize that this is for scoring purposes. The player receives an extra 25 points for every Orc he slays.

That takes care of the attack portion of the handler Fight. Line 324 provides the retaliation attempt. It begins with a GOSUB to Mesprt , since other parts of the handler enter this line with messages to show. It prints a message of its own, message 27, which exclaims, "THE HIDEOUS MONSTER LEAPS AT YOUR THROAT!"

Then the probabilities are calculated for the success or failure of the creature's retaliation. As before, the variable *X* is set to a value from 0 to 100. If *X* is less than 30 (a 30 percent chance), then the creature has been victorious and the adventurer is slain. If so, message 29 is called up and displayed, lamenting, "IT FINISHES YOU OFF!!" At this point, program control is vectored to line 580, which is the routine Resur. This handler (or sub-handler) arranges for the player's re-entry into the game.

There is a 70 percent chance, though, that the creature's retaliation does not succeed. In this case, message 28 is issued, relating a nervous, "SOMEHOW YOU FEND IT OFF!!" All is well, as the Executive is re-entered and the player gets a chance to catch his breath before typing another "KILL" or "SLAY."

THE PLAYER'S RESURRECTION

Even the most experienced adventurer gets eaten once in awhile. To provide a fair second chance, the handler Resur brings the player back for more. There is a cost, of course, to his score. (This prevents players from being reckless instead of clever.)

Resur is given in Fig. 7-3. There are a handful of tasks for it to handle:

```
   NAME:    RESUR

   TYPE:    HANDLER

   FUNCTION:   RESURRECTION OF SLAIN

               PLAYER


580 CT(3)=CT(3)+1:B=35:GOSUB1100:OB(9,
1)=2:FORI=1TO12:IFOB(I,1)=21THENOB(I,1
)=CT(0):NEXT:ELSENEXT
582 CT(0)=1:CT(2)=0:GOTO100
```

Fig. 7-3. Handler Resur.

- To keep track of the number of deaths for later scoring
- To inform the player of his situation
- To make certain the player can get a torch for his next venture
- To empty his carry-sack into the room where he died
- To move the player back to home base
- To update his inventory load total to zero

Resur begins by recording this death in the variable CT(3). Later on, when the score is computed, CT(3) is consulted, and the total score is docked by 20 points per death.

Next, message 35 is printed, to inform the player of his dire situation. It reads, "WELL, FINE ADVENTURER! YOU ARE IN A REAL JAM! FORTUNATELY, WE CAN BRING YOU BACK! . . . POOF!! . . ." At this point you may wish to insert some sort of FOR-NEXT loop in the handler simply for delay. It might help support the illusion of great effort being taken to reassemble the fallen adventurer.

The next step is to make sure the player, once resurrected, is able to re-enter the basement. When he returns, he is outside, above ground. He needs a torch to travel underground—and he dropped his torch "down there, somewhere!" The only fair thing to do is to drop a torch somewhere in reach so that he can return to the basement and reclaim his treasures. Resur takes the torch, which is object 9, and moves it to room 2, above ground. The player can find it easily up there.

Now Resur must steal everything in the adventurer's carry-sack. If he died in a given room, by all rights his possessions should have fallen on the floor there! A FOR-NEXT loop scans through the list of portable objects (of which there are twelve; the others are creatures). Any objects with location number 21 are in the player's possession. Each such object is transferred to the room where the player lies dead, as determined by CT(0).

The last two steps occur in line 582. By setting CT(0) to 1, the player is whisked out of the basement and dropped at home base, or room 1, the excavation pit. Also, since the player no longer is carrying anything, CT(2) must be set to zero. CT(2) is used by the Take handler to determine if the player is carrying too much. At this point, it is reset.

BOMBING THE ENEMY

We have one final handler to examine that relates to the adventurer fighting for his life. Of the four passive creatures, remember, there are two that are impervious to the player's hasty ax swings. Both the White Spider and the Nameless Terror cannot be killed by an ax attack; but they retaliate! Woe to the adventurer who is trapped in the Oracle Room with nothing but his ax. His only escape is to teleport out, for the Nameless Terror guards the only exit, and the Terror laughs at axes.

Fortunately, there is a weapon that kills either of these two hardy beasts. It is the Enchanted Grenade, also known as object 12. If this magic bomb is thrown at one of these tougher creatures, it detonates and blows the beast away in an ethereal burst of light. The grenade cannot operate or explode against any other object; so the adventurer is bound to waste a turn or two trying to blast down a locked door with it.

The handler that controls the operation of the Enchanted Grenade is called Bomb (naturally), and it is given for you in Fig. 7-4. There are two ways in which Bomb is activated. The first was mentioned in the previous chapter when I described the handler Drop. One of the questions that Drop asks is, "Is the adventurer dropping the grenade?" If so, Drop relinquishes the whole matter to Bomb. Since there are basically two forms of the command that invokes Drop, you have two commands right from the start that can activate Bomb; these are "DROP GRENADE" and "THROW GRENADE."

The adventurer can be more specific, however. In the word table, there are two keywords that specifically request the activation of handler 17 (Bomb). These two words are "BOMB" and "BLOW." As you'll see, the handler ignores word 2 of the command altogether.

```
NAME:     BOMB

TYPE:     HANDLER

FUNCTION:    OPERATION OF ENCHANTED

             GRENADE


540 IFOB(12,1)<>21THENB=20:GOTO544:ELS
EOB(12,1)=CT(0):CT(2)=CT(2)-1:FORK=15T
016:IFOB(K,1)<>CT(0)THENNEXTK:B=21:GOT
0544:ELSEOB(K,1)=0:A=1:GOSUB1200:GOSUB
1220:B=22
544 GOSUB1100:GOTO104
```

Fig. 7-4. Handler Bomb.

The player can enter "BOMB CREATURE" or "BLOW UP SPIDER," or whatever, and the program still understands the player's intention.

The handler Bomb must determine the following factors before following through with a grenade explosion:

- Does the adventurer have the grenade?
- Is one of the tough creatures nearby?

Bomb begins by checking to see if the player is bluffing. Does he have a grenade to throw? The Enchanted Grenade is object 12. If the player possesses it, then the variable OB(12,1) equals 21, the location number of the carry-sack. If this is not true, Mesprt is called and prints message 20, "YOU HAVE NO BOMB!"

Assuming that the adventurer does carry the grenade, the handler goes ahead and drops it on the floor. This is done in two steps. First, the grenade is transferred from the carry-sack to the present room by setting OB(1,1) equal to the value in CT(0). Second, since the player's carry-sack is now a bit lighter, this fact must be noted. The variable CT(2), the inventory total, is decremented by one.

The next question is whether or not a legitimate target is within range. The only two creatures whose presence trigger the grenade are the Spider and the Terror, Objects 15 and 16. A short FOR-

NEXT loop checks these two to see if either is in the room, by comparing their locations to CT(0). If neither is in the room, the grenade fizzles. Message 21 is printed, which announces, "THE GRENADE FALLS TO THE FLOOR AND NOTHING HAPPENS."

(Note, additionally, that the grenade is never "used up." Whether or not it explodes, it ends up still lying on the floor. It can be picked up and used later against the other of the two tough creatures. Don't say I never gave you anything!)

If one of the two creatures is in the room, the handler destroys it. The first step is simple. By setting OB(K,1) equal to zero, the creature is banished to the nonexistent room 0, from which it never returns. The real task, though, is to adjust the associated entry in the obstacle list so that the guarded doorway can be declared passable. This is handled just as it is in the handler Fight. The variable *A* is set to one, to indicate the obstacle type, i.e., a creature, and the subroutine Ckobs is called. Ckobs searches the obstacle list and finds the proper entry. The handler calls Revobs, which toggles the obstacle status from unpassable to passable.

The final job is to print a message notifying the adventurer of his triumph. Message 22 fills the bill: "THE GRENADE EXPLODES IN A SILENT FLASH OF WEIRD BLUE LIGHT . . . AND THE CREATURE IS GONE!"

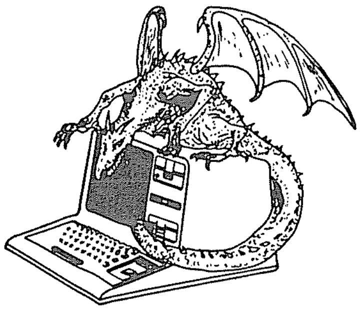
The adventurer is now ready for anything. Axe in one hand, grenade in the other, he can face any beast—be it Orc, Mantis, or Terror. All he needs to know is which weapon works on which creature!

SAFE AND SOUND

The past few chapters have covered, in detail, the major elements of an adventure program. The player can roam through the scenario, looking at each room, just enjoying the view. The player can also affect his environment, swinging doors shut, carrying objects, and so on. And the player can do battle, fighting off beasts with special weapons, winning or sometimes losing.

There are quite a few other commands the adventure program recognizes; however, they are auxiliary to the action of the game. For instance, the player may want to know his score, or perhaps he wishes to examine his carry-sack. These auxiliary commands are described in the next chapter.

Chapter 8



Auxiliary Commands

Most adventure program commands control the motion or action of the adventurer within the scenario. He moves, he opens doors, he takes articles, he fights monsters. Quite a few commands, however, are actually instructions to the program itself, controlling how the game is played or requesting certain information. These are *auxiliary commands.*

There are seven remaining handlers that support these extra commands. Two of the seven supply information about the surrounding scenario. Two more provide the score, one of which results in game termination. Two more make use of the tape recorder supplied with the TRS-80 to store the present status of the game for later retrieval, so that the player may stay long in the basement. The final handler responds to certain inputs with one-liners, strictly for effect.

TAKE A SECOND LOOK

As the adventurer travels from room to room, the program keeps track of which rooms he has been to before and which rooms are new to him. Based on this information, a room is described in one of two ways: with a long paragraph upon first visit, or with a short room title upon subsequent visits. This saves eye strain and boredom from seeing the same long paragraph displayed every time a room is re-entered.

The only problem with this helpful little feature is that players forget. The player is bound to walk into the Cobweb Room and forget what it looks like. Where are the doors? Are there any dangerous cliffs to avoid?

For this reason the adventure program must have the command LOOK. With this command the long description of the room is repeated so that entrances and exits may be clearly seen.

Figure 8-1 gives the handler called Look. It is handler 10 and is invoked only by the command word LOOK. It performs the following two functions:

- Describe the room in detail
- List all nearby objects

Really, this is not so different than what is done by the description subsection of Executive each time a room is entered. All that Look does is ignore the stored information concerning whether or not a room has been visited.

Two subroutines come into play in Look. The first is Viewrm, which prints either the long or short description of a room, based on the value stored in variable *C*. If *C* equals zero, Viewrm prints the long room description. (Look sets *C* to zero and calls line 1160.) Note, though, that Viewrm performs the necessary checks concerning visibility. If the player is below ground and he does not have the torch, Viewrm refuses to give any description and warns the player, "IT IS PITCH DARK! YOU MAY FALL INTO A PIT!"

The other subroutine is Listob at line 1140. Listob runs through the entire list of objects, finds all of the objects that reside in the room, and describes these for the player. Again, if it is too dark, the description does not occur. In this case, however, no warning message is issued.

It may be desirable to review the workings both of Listob and Viewrm. These are described in the chapter detailing the operation of Executive.

TAKING STOCK

Often it is helpful to know what one possesses. If the player

```
NAME:    LOOK

TYPE:    HANDLER

FUNCTION:    LONG DESCRIPTION OF ROOM

380 C=0:GOSUB1160:GOSUB1140:GOTO104
```

Fig. 8-1. Handler Look.

```
        NAME:    INVEN

        TYPE:    HANDLER

        FUNCTION:   LISTING OF OBJECTS

                    CARRIED


340 B=18:GOSUB1100:FORJ=1TO16:IFOB(J,
1)<>21THENNEXTJ:GOTO104:ELSEA=4:B=J:G
OSUB1040:READB$,B$:PRINTB$:NEXTJ:GOTO
104
```

Fig. 8-2. Handler Inven.

encounters the angry Orc, he may or may not remember whether or not he has the Axe among his possessions. Or, upon finding a treasure, he may be told that he is carrying too much. So he must make a decision concerning what to keep and what to drop.

Handler 8 provides an easy way to rummage through the adventurer's carry-sack. It is called Inven and is given in Fig. 8-2.

Basically, Inven takes an inventory of everything the player presently carries. Much like the subroutine Listob, this handler scans the object status array in search of any objects that reside in the carry-sack. These objects are then described by a short-form name.

Inven begins by a preliminary message, message 18, which states, "YOU HAVE THE FOLLOWING:" Then a FOR-NEXT loop is established, and each object location is compared to 21, the carry-sack location. For each object that matches location 21, the short-form name is accessed and displayed.

Every object has two descriptions: a long one about 64 characters maximum and a short one that is one or two words long. These are stored as pairs, one pair to a line, in the object description-block among the DATA statements. For each object to be listed, Inven must find the proper line and short name.

Inven uses the subroutine Access at line 1040 to find the short name. Access expects to see variable *A* equal to the data block number, and variable *B* equal to the row number in that block. The object-description block is numbered 4; so *A* is set accordingly. Since J equals the object number, and the descriptions are stored in rows by that number, Inven sets *B* equal to *J*. When Access is finished, Inven begins reading data and finds the long and short object-descriptions immediately.

Inven only cares about the second of the two descriptions, the shorter one. So the statement "READ B$,B$" results in the short name being stored in B$. The handler prints this name and then loops back. This process continues until all objects have been checked to see if they are being carried.

It may be a bit inconsistent, but note that the player can take inventory at any time and still be able to tell what's in his sack—even in pitch darkness. (Maybe he can identify the objects by size and shape!)

DRAGONS 10, HEROES 0

As the adventurer makes his way from danger to danger, he needs to know how well he is doing—either as an incentive to go on or as a warning to quit while he is still alive. To provide this valuable service, the player can type, "SCORE," and see his progress or lack thereof.

Basements and Beasties uses a very simple scoring algorithm as follows:

- 5 points for every room visited
- 10 points for every treasure at home base
- 20 points for every passive creature killed
- 25 points for every tenacious creature (Orcs) killed
- −20 points for every death of the adventurer

Figure 8-3 gives the code for two portions of the program. Line 420 is the handler Score, which is invoked by entering that same word at the Keyboard. All Score really does is call a subroutine named Points, which is also shown in Fig. 8-3. This was done so that other handlers can make use of Points, notably the handler that ends the present game.

Points begins by printing a preliminary line, message 30, which reads, "YOUR SCORE IS:" Then the contents of CT(4) is dumped into *A*. Remember from the previous chapter that CT(4) keeps track of how many tenacious creatures (Orcs) have been killed; CT(4) is incremented by 25 for every slain Orc. The variable *A* begins with a sizable positive bias if several Orcs are slain in the game.

Next, Points checks the status of each room and awards 5 points for each room that the adventurer has visited. In the room status array RM(x), digit 1 of the integer is a one, if the room was visited, and zero if not. Points isolates digit 1 in two steps: first, by

```
NAME:    SCORE

TYPE:    HANDLER

FUNCTION:   DISPLAY OF CURRENT POINTS

420 GOSUB1240:GOTO104

  NAME:    POINTS

  TYPE:    SUBROUTINE

  INPUT:   NONE

  OUTPUT: THE PRESENT SCORE IS

           CALCULATED AND DISPLAYED

1240 B=30:GOSUB1100:A=CT(4):FORI=1TO20
:IFRIGHT$(STR$(RM(I)),1)="1"THENA=A+5
1242 NEXTI:FORI=1TO8:IFOB(I,1)=1THENA=
A+10
1244 NEXTI:FORI=13TO16:IFOB(I,1)=0THEN
A=A+20
1246 NEXTI:A=A-CT(3)*20:PRINTA:PRINTCT
(1);"STEPS":RETURN
```

Fig. 8-3. Handler Score and the related subroutine Points.

converting the integer into a string with the STR$ function, then by selecting the rightmost character of the new string using the RIGHT$ function. Points performs this analysis for each room, looping from 1 to 20. Each time a one is found, the score increases by five points.

The treasures are tallied next. In order for any treasure to count towards the player's score, it must be safely dropped in room 1, the home base location. Objects 1 through 8 are treasures; so Points checks the location of these special items. Looping through the object status array, a 10-point award is given for each treasure with location number 1.

Now the victorious fighter is shown honor. The hero faces four passive creatures and any number of tenacious Orcs. The Orcs were taken into account at the start of the subroutine; the slain passive beasts are now evaluated. When such a creature is killed, it is removed from the scenario by banishing it to the nonexistent room zero. Thus, Points can easily find out how many passive enemies were killed by checking their location. Points sets up a loop to check objects 13 to 16, which are the regular creatures. Each creature that resides in room 0 earns the adventurer an extra 20 points.

Finally, the player is docked 20 points for each time he fell into a pit, burned in flames, or was eaten. The variable CT(3) keeps track of these failures, and it is multiplied by a factor of 20 and subtracted from the score.

Now Points displays the results. Two pieces of information are printed on the screen. First, the actual score is shown by printing the value of variable *A*. Second, the number of steps taken so far is displayed. The variable CT(1) is used by Executive to accumulate the steps taken; so Points prints its value.

Basically, the player is given two measures of his effectiveness as an adventurer. First, he has a raw score. This score can be as high as 260 if the player is careful not to get killed and can be much higher if chance sends him a few bonus Orcs to vanquish at 25 points a shot. Second, he has an efficiency standard.

The avid adventure gamer first strives for a high raw score. Once he has that in hand, he replays the game for speed and the least number of steps.

WHEN ALL ELSE FAILS

The player always has the option of ending the game if he is either too bored or too frustrated to continue. Of course, he can do this by inelegantly punching BREAK with his thumb. This is the quickest way to call it quits, but just for the sake of style Basements and Beasties includes a QUIT command.

When the player types "QUIT," he invokes handler 14, called Quit, which is given in Fig. 8-4. Quit performs three simple tasks. It issues a sign-off message. This is message 31 in the message block, and it reads, "DO VISIT THE BASEMENT AGAIN!" Whether the player wishes to do so or not is his problem.

Second, Quit displays the adventurer's final raw score and the number of steps taken throughout the game. You have already seen how this is done, using the subroutine Points. Quit simply executes a GOSUB to 1240, and this task is taken care of.

Finally, Quit terminates the program for good. The END statement allows no continuing via a Cont command, so the player shouldn't say "QUIT" until he really means it.

```
NAME:     QUIT

TYPE:     HANDLER

FUNCTION:    GAME TERMINATION

480 B=31:GOSUB1100:GOSUB1240:END
```

Fig. 8-4. Handler Quit.

SAVING THE GAME ON TAPE

Sometimes the adventurer has to quit when he doesn't want to. Dinner won't go away just because he is trapped in the Maze with an Orc at his throat. Even adventurers need sleep now and then. To lend some semblance of normalcy to the player's life, it is helpful to provide the option of saving the game, as is, on tape for resumption later.

Basements and Beasties has two handlers to support tapebased adventure interruption. The first stores all crucial variables on tape; the second recalls them from tape. The input key word SAVE invokes the first handler, and RESTORE activates the second one.

Now, what really needs to be saved on tape to preserve the present status of an adventure program? The handler Save writes the following variables out to the cassette port.

- The present room location of the player
- The number of steps taken so far
- The inventory total count
- The present number of player deaths
- The present number of Orcs killed
- The status of all rooms
- The status of all objects

In writing this handler a rather difficult trade-off presented itself, as we'll see in a moment. The trade-off revolves around the operation of the BASIC statements for tape data files, which are PRINT#-1 and INPUT#-1. These two statements can be used in a variety of ways, some more and some less efficient.

Consider that data is saved on tape, using the PRINT#-1 statement, in *bursts* of up to 255 bytes of data. Each burst of data is preceded by a synchronization leader signal of about five seconds. This means that the most time taken up in data file tapes is due to the leader signals. Clearly, if you want efficient tape storage, you must keep these leaders to a minimum.

Of course, since there is a five-second leader to each data burst, your goal should be to store the desired data in as few bursts as possible. A burst ranges from 1 to 255 bytes in length. To minimize leader time, the Save handler records as many variables as possible in each burst.

Look at Fig. 8-5, though, for two examples of how to save a series of variables on tape. In the first example there is an array A(x) with ten elements to store. A FOR-NEXT loop is set up to store each of the ten elements using a repeated PRINT#-1. The problem is this: each time the statement PRINT#-1 is reexecuted, a new leader signal is recorded and a new data burst initiated. The result is that method 1 produces a long tape file consisting of ten separate bursts, each just over five seconds long. One simple array takes almost a minute to store—and to reload!

Now look at the second example in Fig. 8-5. This time, instead of looping to save each array element, all ten elements are explicitly specified with commas as separators. All ten are stored with only one executed PRINT#-1 statement. The result is that all ten array elements are stored in one single burst with only one leader. (Ten integer variables at about five bytes a piece are only 50 bytes.) Instead of a minute to save or reload the array as before, it now only takes five or six seconds. What a difference!

Here's the catch—the handler Save has a lot of variables to record. The second method is fast and efficient with regard to tape, but wasteful of BASIC code in memory. Imagine that all 20 elements of the room status Array, all ten of the obstacle list, and so on, are all spelled out explicitly! This takes up quite a few lines and a lot of bytes in memory.

As usual, a sort of compromise can be struck, which is neither as fast as possible, nor as lengthy as the extra speed requires. The final version is shown in Fig. 8-6; this version saves all of the crucial variables in nine densely packed bursts, for a total record length of about 48 seconds. If it is done using many FOR-NEXT loops and no comma separators, it takes 52 inefficient bursts, or well over four minutes.

```
20 FOR I=1 TO 10: PRINT#-1,A(I): NEXT
22 PRINT#-1,A(1),A(2),A(3),A(4),A(5),
A(6),A(7),A(8),A(9),A(10)
```

Fig. 8-5. Two ways to save a variable array.

```
NAME:     SAVE

TYPE:     HANDLER

FUNCTION:    SAVING GAME PARAMETERS ON

             TAPE

500 B=19:GOSUB1100:INPUTA:FORI=0TO8:PR
INT#-1,OB(I,0),OB(I,1),OB(I+8,0),OB(I+
8,1),RM(I),RM(I+8),RM(I+12),BK(I),BK(I
+2),CT(I):NEXT:GOTO104
```

Fig. 8-6. Handler save.

The handler Save first prompts the player with message 19, which instructs, "PREPARE TAPE RECORDER AND HIT (ENTER)." The subsequent statement, INPUT A, holds the handler in suspension until ENTER is pressed, to allow the recorder to be loaded and started.

As soon as ENTER is hit, the handler performs a loop from 0 to 8 (which adds up to nine loops total). Let's look at how each array is saved in this loop. You'll see that some variables are actually saved twice, but this redundancy actually helps keep the loops simpler and more efficient.

The first array to be saved is the object status array. The first two items in the list after the PRINT#-1 statement save OB(0,0) to OB(8,0) and OB(0,1) to OB(8,1) in the various bursts.

What about the rest of the array? The next two list items add eight to the value of the loop counter in referencing the array elements to be saved. Elements OB(8,0) to OB(16,0) and OB(8,1) to OB(16,1) are saved. Now, OB(8,0) and OB(8,1) are saved twice by this scheme, but as it turns out that small redundancy is more than compensated by the density of the bursts.

Next to be saved in each burst are the elements of the room status array. One item of the list saves RM(0) to RM(8). The next adds eight to the loop counter, so RM(8) to RM(16) are recorded. The next item adds 12 to the loop counter, such that RM(12) to RM(20) are stored away. Note, again,that a handful of individual array elements are repeated, but this is compensated. The idea is to make one FOR-NEXT loop cover everything, to keep the number of loops minimal.

Next in line is the obstacle list. The first applicable item saves BK(0) to BK(8). The next adds two to the counter, so BK(2) to BK(10) are stored. Again, there is redundancy, but also simplicity.

The last array to be saved is CT(X), the general-purpose record-keeping variables. The most crucial of these are CT(0) through CT(4), but Save keeps everything up to CT(8) at any rate.

Save records all of the major variables of Basements and Beasties in only nine data bursts. Each burst is still somewhat inefficient, since a maximum of about 80 bytes is stored each time—about a third of what is theoretically possible. The only way to increase this efficiency is to shorten the loop and lengthen the list of explicitly stated array elements. I wrote one version that ran in only five loops—but it took twice the memory space in the program!

Corresponding to the handler that saves these variables on tape is one that recalls them to resume a game. The handler Restore is given in Fig. 8-7.

I don't need to go into detail on Restore, since in most respects it is identical to Save. The only difference is that instead of the tape output statement PRINT#-1, there is the tape input statement INPUT#-1. The close similarity between the two handlers assures that all variables are properly loaded. The state of the game before Save is identical to the state after restore.

By the way, you might be wondering if there is a way to save code somehow, since the two handlers are so alike. Can't they share, somehow? Not easily, since the majority of either handler is so closely tied to the input or output statement.

At one point in the development of the program, I attempted to manage this by writing a small routine that *changed* the PRINT#-1

```
NAME:     RESTORE

TYPE:     HANDLER

FUNCTION:    LOADING GAME PARAMETERS

             FROM TAPE
520 B=19:GOSUB1100:INPUTA:FORI=0TO8:IN
PUT#-1,OB(I,0),OB(I,1),OB(I+8,0),OB(I+
8,1),RM(I),RM(I+8),RM(I+12),BK(I),BK(I
+2),CT(I):NEXT:GOTO104
```

Fig. 8-7. Handler Restore.

```
NAME:     LINERS

TYPE:     HANDLER

FUNCTION:    ONE-LINE RESPONSES TO

             KEYWORDS

360 CT(5)=N:GOSUB1000:B=CT(9)*10+CT(8)
:GOSUB1100:GOTO0104
```

Fig. 8-8. Handler Liners.

statement into an INPUT#-1 statement, using the POKE command. (After all, both statements are stored in memory as a specific one-byte code.) However, the small routine kept growing faster than I expected and didn't save enough memory to justify the trouble.

SNAPPY REMARKS

Believe it or not, there is only one handler left that you have not examined. It provides one of the fine points of adventure programming that really adds to the feel of the game. In short, it provides snide remarks to specific inputs.

Figure 8-8 is the handler called Liners. It has one function: to output a specific message in response to a specific command key word.

In order to use Liners, the desired trigger key-words are stored in the word list. Remember that verbs in the word list point to their handler using digits 1 and 2 of their ID number. All trigger key-words have "09" in digits 1 and 2 to invoke handler 9, which is Liners.

Some verbs, though, make use of digits 3 and 4 for special purposes. Direction verbs use these digits to pass a direction number along to the handler Xmove, for example. To keep everything simple, trigger key-words use digits 3 and 4 to pass along a message number, i.e., the number of the message that is printed in response to that input.

At present there are only two trigger key-words in the word list. The first is HELP. In a number of adventure programs that run on larger machines, the command HELP provides a lengthy introduction to the program with an explanation of how to play. Alas, you don't have the memory to waste on such luxuries. Instead, pull the player's leg. The ID number of HELP causes message 37 to be printed, which reads, "YOUR CRIES GO UNHEARD, PITIFUL WRETCH."

The other trigger command is WAIT. In some programs the player can enter this command and expect some situation to change. For instance, if a magic beanstalk is growing, the adventurer can WAIT and the stalk grows before his eyes. Again, Basements and Beasties pokes fun. Message 38 is called, which says, "TIME PASSES . . .."

The operation of the handler Liners is simple. The variable N already contains the ID number of the input word, thanks to the able assistance of Executive. Liners sets variable CT(5) equal to N and calls the subroutine Analyz. This breaks the ID number down into its five digits. Next, Liners needs to isolate the message number embedded in the ID number. Digits 3 and 4 are now stored respectively in CT(8) and CT(9). The expression CT(9)*10+CT(8) recreates the message number, which is stored in variable B. Finally, a call to Mesprt displays the desired message.

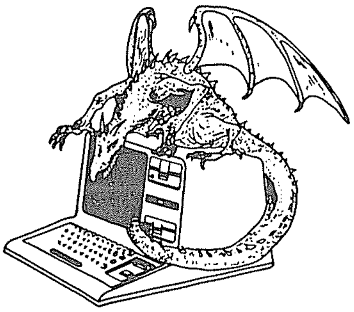
What other sorts of one-liner responses might be added? Sometimes, an adventurer who suspects that the program recognizes a magic word tries one of his own, like ABRACADABRA. That can be placed in the word list and used to trigger the response, "OLD, WORN-OUT MAGIC WORDS HAVE NO EFFECT HERE!" Or, the player standing on a bridge might enter the command JUMP. The program could respond, "HOW HIGH?" You get the idea.

BOY, WAS THAT SIMPLE!

Amazing as it seems, you have covered all the code in Basements and Beasties. You've seen how the program initializes all variables. You've examined Executive and seen it describe rooms and other things. You've followed input key-words from keyboard to word list to handler, with subroutines as supporting cast. You've studied how the adventurer walks about, opening doors, taking treasures, slaying beasts, and getting killed. Finally, you've seen how the program itself is enhanced with special features.

Where does this leave you? With two final tasks. One is to compile all of the preceding bits of code into one complete listing with various tables, annotations, and so on, so that you can load and run Basements and Beasties. That is done in the next chapter. The other is to suggest a number of optional improvements to the adventure program, either to increase speed, efficiency, or capability. That is reserved for the following chapter.

Chapter 9



Basements and Beasties: The Listing

Finally, after much tedium, toil and talk, you have arrived at a complete listing of the program Basements and Beasties. Before you roll up your sleeves and tackle the task of typing it in, a couple of comments are in order.

First, you'll notice that there are no REM statements whatsoever in the listing. (Hopefully, the remarks in the text make remarks in the listing superfluous.) The reason, obviously, is that remarks take up memory space, just what you are trying to conserve.

What may be more discomforting is that there are no spaces, except for lines containing strings to be displayed on the screen. Again, the rationale was memory savings, since BASIC doesn't really need spaces everywhere. This can certainly generate some frustration when it comes time to type! Everything runs together so that variable names are hard to separate from BASIC key words, and so on. Words and expressions can get chopped in half between succeeding lines.

Be careful while you type. If your keyboard has noticeable keybounce, keep an eye on the screen, because errors of this kind are harder to find in these tightly packed program lines. It may even be advisable to read each line before you type it, deciphering the line in order to be prepared for expressions you might mistype.

Finally, you may wish to put spaces into some lines for clarity; go ahead. But if you do, I suggest that you avoid spaces in the word

HANDLER	LINE	HANDLER	LINE
XMOVE	200	READ	400
IMOVE	220	SCORE	420
TAKE	240	SAY	460
DROP	260	QUIT	480
OPEN	280	SAVE	500
CLOSE	300	RESTORE	520
FIGHT	320	BOMB	540
INVEN	340	AARDVARK	560
LINERS	360	RESUR	580
LOOK	380		

Fig. 9-1. Handlers for Basements and Beasties.

SUB- ROUTINE	LINE	SUB- ROUTINE	LINE
ANALYZ	1000	LISTOB	1140
SYNTHE	1020	VIEWRM	1160
ACCESS	1040	DARKCK	1180
GETCOM	1060	CKOBS	1200
IDWORD	1080	REVOBS	1220
MESPRT	1100	POINTS	1240
TRAVEC	1120		

Fig. 9-2. Subroutines for Basements and Beasties.

table. Why? Because in the next chapter, you'll be implementing a machine-language routine that scans the word table at high speed. The routine assumes no spaces in the table, and leaving them out now will prevent having to remove them later.

Prior to the listing itself, there are two lists included for your reference. One lists the handlers and the other the subroutines, in order and by line numbers. This may in part make up for the lack of REM statements with regard to finding specific segments of code.

```
2 CLS:PRINTCHR$(23):PRINT@468,"WELCOM
E TO":PRINT@522,"BASEMENTS & BEASTIES
"
4 CLEAR500:DEFINTA-Z:DIMTX$(4),DA(5),
RM(20),OB(16,1),BK(10),CT(12):FORI=1T
020:READRM(I):NEXT:FORI=1T016:READOB(
I,1),OB(I,0):NEXT:FORI=1T010:READBK(I
):NEXT
6 P=17385: N=1: FORI=5000T09000STEP1000

8 IFI=PEEK(P+2)+PEEK(P+3)*256THENDA(N
)=P:N=N+1:NEXTI:GOTO10:ELSEP=PEEK(P)+
PEEK(P+1)*256:IFP=0THENCLS:PRINT"ERRO
R":END:ELSE8
10 CT(0)=1:CT(12)=RND(10)+10:CLS
100 CT(5)=RM(CT(0)):GOSUB1000:C=CT(6)
:GOSUB1160:GOSUB1180:IFB=0ANDC=0THENC
T(6)=1:GOSUB1020:RM(CT(0))=CT(5):ELSE
IFB=1THENN=RND(100): IFN<20THENB=5:GOS
UB1100:GOT0580
102 GOSUB1140
104 GOT0112
105 INPUTA$
106 GOSUB1060:A$=TX$(2):GOSUB1080
108 CT(5)=N:GOSUB1000:IFCT(10)=0ORN=0
THENB=7:GOSUB1100:GOT0104
110 ONCT(6)+CT(7)*10GOTO200,220,240,2
60,280,300,320,340,360,380,400,420,46
0,480,500,520,540,560,580,600,620,640
,660,680,700
112 IFOB(0,0)=0ANDCT(0)>2THENCT(12)=C
T(12)-1:IFCT(12)<=0THENCT(12)=RND(10)
+10:OB(0,1)=CT(0):OB(0,0)=1:GOTO116:E
LSE105
114 IFCT(0)<3THENOB(0,0)=0:GOTO105:EL
SEOB(0,1)=CT(0)
```

Fig. 9-3. The complete listing for *Basements and Beasties* for the TRS-80 Model III computer. To run the program on the Model I, in line 6 you must change the number 17385 to the number 17129.

```
116 B=42:GOSUB1100:B=RND(100):IFB>75T
HEN105ELSEB=43:GOSUB1100:B=RND(100):I
FB>60THENB=44:GOSUB1100:GOTO580:ELSE1
05
200 D=CT(8)+CT(9)*10-1:FORK=1TO10:CT(
5)=BK(K):GOSUB1000:IFD<>CT(8)ORCT(0)<
>CT(6)+CT(7)*10THENNEXTK:GOTO202:ELSE
IFBK(K)<0THEN202ELSEB=CT(9):GOTO206
202 D=D+1:GOSUB1120:IFA=22THENB=4:GOT
O204:ELSEIFA=23THENB=5:GOTO204:ELSEIF
A=0THENB=6:GOTO206:ELSECT(0)=A:CT(1)=
CT(1)+1:GOTO100
204 GOSUB1100:GOTO580
206 GOSUB1100:GOTO104
220 IFTX$(3)=""THEND=11:GOSUB1120:N=A
*100+10101:GOTO108:ELSEA$=TX$(3):GOTO
106
240 IFCT(2)>=5THENB=36:GOSUB1100:GOTO
104:ELSEA$=TX$(3):GOSUB1080:IFN>9999T
HENB=7:GOTO242:ELSEIFN>12ANDN<17ORN=1
8THENB=40:GOTO242
241 IFN=17THENB=8:GOTO242:ELSEIFOB(N,
1)=21THENB=9:GOTO242:ELSEIFOB(N,1)<>C
T(0)ORN=0THENB=12:GOTO242:ELSEOB(N,1)
=21:B=11:CT(2)=CT(2)+1
242 GOSUB1100:GOTO104
260 A$=TX$(3):GOSUB1080:IFN>9999THENB
=7:GOSUB262:ELSEIFOB(N,1)<>21THENB=10
:GOTO262:ELSEIFN=12THEN540:ELSEOB(N,1
)=CT(0):B=11:CT(2)=CT(2)-1
262 GOSUB1100:GOTO104
280 IFTX$(3)=""THENB=7:GOTO284:ELSEA$
=TX$(3):GOSUB1080:CT(5)=N:GOSUB1000:A
=CT(8):GOSUB1200:IFA=0THENB=12:GOTO28
4:ELSEIFBK(A)<0THENB=13:GOTO284:ELSEI
FOB(11,1)<>21THENB=16:GOTO284:ELSEGOS
UB1220:B=12+CT(9)
284 GOSUB1100:GOTO104
300 IFTX$(3)=""THENB=7:GOTO304:ELSEA$
=TX$(3):GOSUB1080:CT(5)=N:GOSUB1000:A
=CT(8):GOSUB1200:IFA=0THENB=12:GOTO30
4:ELSEIFBK(A)>0THENB=13:GOTO304:ELSEG
OSUB1220:B=17
304 GOSUB1100:GOTO104
320 IFOB(0,0)=1THEN322ELSEFORK=13TO16
```

```
:IFOB(K,1)<>CT(0)THENNEXTK:B=41:GOSUB
1100:GOTO104
322 IFOB(10,1)<>21THENB=23:GOTO326:EL
SEIFK=15ORK=16THENB=24:GOTO324:ELSEX=
RND(100):IFOB(0,0)=1THEN328ELSEIFX>70
THENB=26:GOTO324:ELSEOB(K,1)=0:A=1:GO
SUB1200:GOSUB1220:B=25:GOTO326
324 GOSUB1100:B=27:GOSUB1100:X=RND(10
0):IFX<40THENB=29:GOSUB1100:GOTO580:E
LSEB=28
326 GOSUB1100:GOTO105
328 IFX>70THENB=26:GOSUB1100:GOTO112:
ELSEOB(0,0)=0:OB(0,1)=0:B=25:CT(4)=CT
(4)+25:GOTO326
340 B=18:GOSUB1100:FORJ=1TO16:IFOB(J,
1)<>21THENNEXTJ:GOTO104:ELSEA=4:B=J:G
OSUB1040:READB$,B$:PRINTB$:NEXTJ:GOTO
104
360 CT(5)=N:GOSUB1000:B=CT(9)*10+CT(8
):GOSUB1100:GOTO104
380 C=0:GOSUB1160:GOSUB1140:GOTO104
400 IFCT(0)<>6THENB=32:GOTO402:ELSEB=
33
402 GOSUB1100:GOTO104
420 GOSUB1240:GOTO104
460 IFLEFT$(TX$(3),5)<>"AARDV"THENB=3
4:GOSUB1100:GOTO104:ELSE560
480 B=31:GOSUB1100:GOSUB1240:END
500 B=19:GOSUB1100:INPUTA:FORI=0TO8:P
RINT#-1,OB(I,0),OB(I,1),OB(I+8,0),OB(
I+8,1),RM(I),RM(I+8),RM(I+12),BK(I),B
K(I+2),CT(I):NEXT:GOTO104
520 B=19:GOSUB1100:INPUTA:FORI=0TO8:I
NPUT#-1,OB(I,0),OB(I,1),OB(I+8,0),OB(
I+8,1),RM(I),RM(I+8),RM(I+12),BK(I),B
K(I+2),CT(I):NEXT:GOTO104
540 IFOB(12,1)<>21THENB=20:GOTO544:EL
SEOB(12,1)=CT(0):CT(2)=CT(2)-1:FORK=1
5TO16:IFOB(K,1)<>CT(0)THENNEXTK:B=21:
GOTO544:ELSEOB(K,1)=0:A=1:GOSUB1200:G
OSUB1220:B=22
544 GOSUB1100:GOTO104
560 IFCT(0)=6THENCT(0)=1ELSEIFCT(0)=1
THENCT(0)=6ELSEB=34:GOSUB1100
562 GOTO100
580 CT(3)=CT(3)+1:B=35:GOSUB1100:OB(9
```

Fig. 9-3. Continued from page 135.

```
,1)=2:FORI=1TO12:IFOB(I,1)=21THENOB(I
,1)=CT(0):NEXT:ELSENEXT
582 CT(0)=1:CT(2)=0:GOTO100
1000 FORZ=6TO10:CT(Z)=0:NEXTZ:B$=MID$
(STR$(CT(5)),2):FORZ=1TOLEN(B$):CT(6+
LEN(B$)-Z)=VAL(MID$(B$,Z,1)):NEXTZ:IF
CT(5)<0THENCT(11)=-1:RETURN:ELSECT(11
)=1:RETURN
1020 CT(5)=CT(10)*10000+CT(9)*1000+CT
(8)*100+CT(7)*10+CT(6):CT(5)=CT(5)*CT
(11):RETURN
1040 P=DA(A):IFB=1THEN1042ELSEFORZ=1T
OB-1:P=PEEK(P)+PEEK(P+1)*256:NEXTZ
1042 P=P-1:POKE16640,FIX(P/256):POKE1
6639,P-FIX(P/256)*256:RETURN
1060 FORI=1TOLEN(A$):IFMID$(A$,I,1)<>
" "THENNEXTI:TX$(3)="":TX$(2)=A$:RETU
RN:ELSETX$(3)=MID$(A$,I+1):TX$(2)=LEF
T$(A$,I-1):RETURN
1080 IFLEN(A$)>5THENA$=LEFT$(A$,5)
1082 A=2:B=1:GOSUB1040
1084 READB$,N:IFB$="."ORB$=A$THENRETU
RNELSE1084
1100 A=3:GOSUB1040:READA$:PRINTA$:RET
URN
1120 B=CT(0):A=1:GOSUB1040:FORY=1TOD:
READA:NEXTY:RETURN
1140 GOSUB1180:IFB=1THENRETURN:ELSEA=
4:FORB=1TO16:IFCT(0)<>OB(B,1)THENNEXT
B:RETURN:ELSEGOSUB1040:READTX$(4):PRI
NTTX$(4):NEXTB:RETURN
1160 GOSUB1180:IFB=1THENB=39:GOSUB110
0:RETURN:ELSEA=5:B=CT(0):GOSUB1040:RE
ADTX$(0),TX$(1):IFC=0THENPRINTTX$(0):
RETURN:ELSEPRINTTX$(1):RETURN
1180 IFOB(9,1)<>21ANDCT(0)<>1ANDCT(0)
<>2THENB=1ELSEB=0
1182 RETURN
1200 FORQ=1TO10:CT(5)=BK(Q):GOSUB1000
:IFCT(6)+CT(7)*10<>CT(0)ORCT(9)<>ATHE
NNEXTQ:A=0:ELSEA=Q
1202 RETURN
1220 BK(A)=-BK(A):CT(5)=BK(A):GOSUB10
00:IFCT(10)=1RETURNELSEBK(A-1+CT(10))
=-BK(A-1+CT(10)):RETURN
1240 B=30:GOSUB1100:A=CT(4):FORI=1TO2
```

```
0:IFRIGHT$(STR$(RM(I)),1)="1"THENA=A+
5
1242 NEXTI:FORI=1TO8:IFOB(I,1)=1THENA
=A+10
1244 NEXTI:FORI=13TO16:IFOB(I,1)=0THE
NA=A+20
1246 NEXTI:A=A-CT(3)*20:PRINTA:PRINTC
T(1);"STEPS":RETURN
2000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0
3000 DATA 4,0,7,0,20,0,11,0,5,0,19,0,
7,0,6,0,2,0,3,0,10,0,12,0,4,0,18,0,14
,0,6,0
4000 DATA 22902,2808,23306,3712,23404
,3011,11104,11118,11714,11306,0
5000 DATA1,2,2,1,1,1,1,1,0,3,9
5002 DATA2,2,2,2,2,1,1,2,0,8,9
5004 DATA0,0,4,10,0,0,0,0,1,0,8
5006 DATA0,5,0,0,11,0,3,0,0,0,4
5008 DATA0,0,0,0,0,4,0,0,0,0,5
5010 DATA0,0,0,12,0,0,0,0,0,23,3
5012 DATA0,0,0,14,0,0,0,0,0,0,3
5014 DATA0,0,0,0,14,0,0,0,2,0,8
5016 DATA9,0,16,15,9,0,0,9,0,0,7
5018 DATA23,23,23,16,17,17,17,3,0,17,
4
5020 DATA4,0,0,0,0,0,0,0,0,0,0
5022 DATA0,0,13,0,18,0,0,6,0,0,7
5024 DATA0,0,0,0,0,0,12,0,0,0,6
5026 DATA8,0,0,0,19,0,0,7,0,19,4
5028 DATA15,0,15,0,15,16,9,0,0,0,0
5030 DATA15,16,16,0,16,0,10,9,0,0,1
5032 DATA18,18,18,18,18,18,18,18,18,1
8,0
5034 DATA12,19,0,0,0,0,0,0,0,0,0
5036 DATA14,0,0,0,0,18,0,0,14,20,9
5038 DATA22,22,22,22,22,22,22,22,19,2
2,8
6000 DATAJEWEL,1,CROWN,1,GOLDE,2,CUBE
,2,DIAMO,3,BEETL,3,SILVE,4,BELT,4,PLA
TI,5,RING,5,ONYX,6,COIN,7,HOURG,8
6001 DATATORCH,9,AXE,10,KEY,11,GRENA,
12,MANTI,13,IGUAN,14,SPIDE,15,NAMEL,1
6,TERRO,16,ORC,18
6002 DATAOOZE,17,DESKS,17,CABIN,17,BO
DIE,17,COKE,17,MACHI,17,COBWE,17,CASE
```

Fig. 9-3. Continued from page 137.

```
S,17,COFFI,17,DOOR,317,GRATE,217,U,10
901,D,11001,N,10101,NE,10201,E,10301,
SE,10401,S,10501,SW,10601,W,10701,NW,
10801
6003 DATANORTH,10101,SOUTH,10501,EAST
,10301,WEST,10701,UP,10901,DOWN,11001
,SCORE,10012,QUIT,10014,KILL,10007,FI
GHT,10007,SLAY,10007,BLOW,10017,BOMB,
10017
6004 DATAWAIT,13809,HELP,13709,READ,1
0011,SAY,10013,LOCK,10006,UNLOC,10005
,OPEN,10005,SHUT,10006,CLOSE,10006,LO
OK,10010,INVEN,10008
6005 DATATAKE,10003,DROP,10004,THROW,
10004,STEAL,10003,IN,10002,OUT,10002,
GO,10002,ENTER,10002,EXIT,10002,SAVE,
10015,RESTO,10016,AARDV,10018,.,0
7000 DATA "THE CREATURE WILL NOT LET
YOU PASS!"
7001 DATA"THE GRATE IS CLOSED AND LOC
KED!"
7002 DATA"THE DOOR IS TIGHTLY SHUT AN
D LOCKED."
7003 DATA"YOU BURN IN THE FLAMES!"
7004 DATA"YOU FALL TO YOUR DOOM..."
7005 DATA"YOU CANT GO THAT WAY"
7006 DATA"WHAT DID YOU SAY?"
7007 DATA"YOU TRY UNSUCCESSFULLY...IM
MOVABLE!"
7008 DATA"YOU ALREADY HAVE IT!"
7009 DATA"YOU DONT HAVE IT!"
7010 DATA"OKAY."
7011 DATA"I SEE NOTHING OF THE SORT H
ERE."
7012 DATA"YOU DONT NEED TO."
7013 DATA"WITH A CREAK, THE GRATE FAL
LS OPEN."
7014 DATA"THE DOOR SWINGS OPEN WIDE."

7015 DATA"YOU HAVE NO KEY!"
7016 DATA"IT SLAMS SHUT AND THE LOCK
CATCHES."
7017 DATA"YOU HAVE THE FOLLOWING:"
7018 DATA"PREPARE TAPE RECORDER AND H
IT <ENTER>."
```

```
7019 DATA"YOU HAVE NO BOMB!"
7020 DATA"THE GRENADE FALLS TO THE FL
OOR AND NOTHING HAPPENS."
7021 DATA"THE GRENADE EXPLODES IN A S
ILENT FLASH OF WEIRD BLUE
LIGHT...AND THE CREATURE IS GONE!"
7022 DATA"WITH WHAT WEAPON?"
7023 DATA"YOUR AXE SWINGS ARE DYNAMIC
...BUT INEFFECTIVE!"
7024 DATA"YOUR MAGIC AXE CONNECTS! TH
E CREATURE VANISHES IN
A PUFF OF FOUL SMOKE!"
7025 DATA"MISSED IT! FIE!"
7026 DATA"THE HIDEOUS MONSTER LEAPS A
T YOUR THROAT!"
7027 DATA"SOMEHOW YOU FEND IT OFF!"
7028 DATA"IT FINISHES YOU OFF!!"
7029 DATA"YOUR SCORE IS:"
7030 DATA"DO VISIT THE BASEMENT AGAIN
!"
7031 DATA"NOTHING HERE TO READ...HOW
DULL!"
7032 DATA"THE DANGER HERE
IS PRETTY THICK,
BUT SAY <AARDVARK>;
YOULL GET OUT QUICK!"
7033 DATA"NOTHING HAPPENS."
7034 DATA"WELL, FINE ADVENTURER! YOU
ARE IN A REAL JAM!
FORTUNATELY, WE CAN BRING YOU BACK!
         ...POOF!!..."
7035 DATA"YOUR ARMS ARE FULL...YOU CA
N CARRY NO MORE."
7036 DATA"YOUR CRIES GO UNHEARD, PITI
FUL WRETCH."
7037 DATA"TIME PASSES..."
7038 DATA"IT IS PITCH DARK! YOU MAY F
ALL INTO A PIT!"
7039 DATA"YOU MANIFEST SOME PRETTY SU
ICIDAL TENDENCIES, FELLA!"
7040 DATA"SAVE YOUR STAMINA, TURKEY!
I SEE NO REAL THREAT!"
7041 DATA"THERE IS AN ANGRY ORC NEARB
Y!"
7042 DATA"HE SWINGS OUT AT YOU WITH A
  BLACK SCIMITAR!"
```

Fig. 9-3. Continued from page 139.

```
7043 DATA"YOU ARE SLASHED IN PIECES."
8000 DATA "THERE IS A CROWN OF JEWELS
 HERE!","JEWELED CROWN"
8001 DATA"THERE IS A GOLDEN CUBE HERE
!","GOLDEN CUBE"
8002 DATA"THERE IS A DIAMOND HERE CAR
VED LIKE A BEETLE!","DIAMOND BEETLE"
8003 DATA"THERE IS A FINE SILVER BELT
 HERE!","SILVER BELT"
8004 DATA"THERE IS A RING HERE OF PUR
E PLATINUM!","PLATINUM RING"
8005 DATA"THERE IS A POLISHED ONYX HE
RE!","ONYX"
8006 DATA"THERE IS A COIN HERE WORTH
MILLIONS!","COIN"
8007 DATA"THERE IS AN ANCIENT HOURGLA
SS HERE!","HOURGLASS"
8008 DATA"THERE IS A BURNING TORCH HE
RE.",TORCH
8009 DATA"THERE IS A HEFTY MAGIC AXE
HERE.",AXE
8010 DATA"THERE IS A LARGE KEY HERE."
,KEY
8011 DATA"THERE IS AN ENCHANTED GRENA
DE HERE.",GRENADE
8012 DATA"A GIANT MANTIS CROUCHES NEA
RBY, READY TO POUNCE!"
8013 DATA"A HUGE IGUANA PACES RESTLES
SLY NEARBY, KEEPING AN
EYE ON YOU!"
8014 DATA"A GIANT WHITE SPIDER, MANDI
BLES TWITCHING, TOWERS
ABOVE YOU!"
8015 DATA"THE NAMELESS TERROR ARISES
FROM A PIT, BLOCKING
YOUR RETREAT WITH SLIMY TENTACLES!!"
9000 DATA "YOU STAND AT THE BOTTOM OF
 A LARGE PIT. AT YOUR FEET IS A
NARROW HOLE JUST WIDE ENOUGH TO CRAWL
 INTO.","BOTTOM OF PIT"
9001 DATA"HERE ARE THE RUINS OF AN AN
CIENT TROLL-CASTLE. NEARBY
IS A GRATE LEADING DOWN INTO DARKNESS
...","RUINS"
9002 DATA"THIS WAS APPARENTLY ONCE A
WEAPONS ROOM, THOUGH THE
```

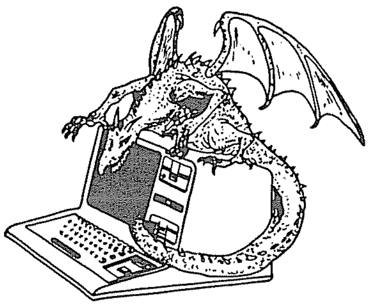
```
CASES ARE ALL EMPTY NOW. THERES A HOL
E IN THE ROOF, AN ARCHWAY
TO THE EAST, AND A JAGGED HOLE IN THE
 SOUTHEAST WALL.","WEAPONS ROOM"
9003 DATA"THE SIGNS OF A GREAT BATTLE
 BETWEEN TROLLS AND TERRIBLE
BEAST-MEN ARE EVIDENT...FROM THE LOOK
S OF IT, THE TROLLS LOST.
BODIES ARE EVERYWHERE. THERE IS A JAG
GED HOLE TO THE WEST, A
HALL NORTHEAST, AND A SOUTH DOOR.","L
OST BATTLE"
9004 DATA"THE WALLS ARE LINED WITH CO
FFIN CASES...THIS IS
THE TROLL CEMETERY, IT SEEMS. A SOUTH
WEST DOOR LEADS OUT.","TOMB ROOM"
9005 DATA"THIS IS A SMALL, DARK ROOM
SMELLING OF MAGIC. THE
ORACLE HAS LEFT A MESSAGE ON THE WALL
. THERES A SOUTHEAST DOOR
AND A LARGE PIT NEAR THE DOOR.","ORAC
LE ROOM"
9006 DATA"AT LAST! THE TREASURE VAULT
! WHAT A SHAME THAT SO
MUCH OF THE ORIGINAL WEALTH HAS BEEN
REMOVED! THERE IS A
SOUTHEAST DOOR OUT.","TREASURE VAULT"

9007 DATA"THIS WAS ONCE THE MAIN GUAR
DPOST TO THE UNDERGROUND
KINGDOM OF THE TROLLS. THERE IS AN EN
TRANCE-GRATE SET IN THE
ROOF AND A SOUTH EXIT DOOR.","GUARD P
OST"
9008 DATA"YOU ARE LOST IN A MAZE!","Y
OU ARE LOST IN A MAZE!"
9009 DATA"YOU WALK ALONG A NARROW LED
GE RUNNING NORTHWEST AND SOUTHEAST.
TO THE WEST IS A RAPID STREAM FAR BEL
OW, AND TO THE EAST IS A
BOTTOMLESS CHASM!","NARROW LEDGE"
9010 DATA"THIS IS A SMALL PRISON CELL
. THROUGH THE BARS, YOU CAN
SEE A NICE OFFICE...UNREACHABLE. THER
ES A NORTH DOOR.","CELL"
9011 DATA"HERE IS A BUSINESS OFFICE,
```

Fig. 9-3. Continued from page 141.

```
WITH EMPTY, RANSACKED DESKS AND
CABINETS. A BARRED WINDOW IN THE WALL
 SHOWS A SMALL PRISON CELL
OF SOME SORT. THERE ARE TWO DOORS, TO
 THE NORTHWEST AND EAST,
AND A ROCKY HOLE IN THE SOUTH WALL.",
"OFFICE"
9012 DATA"THIS IS THE LUNCH ROOM, COM
PLETE WITH COKE MACHINE...
EMPTY, UNFORTUNATELY. THERE IS A DOOR
 TO THE WEST.","LUNCH ROOM"
9013 DATA"WHAT A CREEPY PLACE! THERE
ARE COBWEBS EVERYWHERE! A
DOOR LEADS NORTH, A HALL GOES NORTHWE
ST, AND THERE IS A HOLE
IN THE FLOOR.","COBWEB ROOM"
9014 DATA"YOU ARE LOST IN A MAZE!","Y
OU ARE LOST IN A MAZE!"
9015 DATA"YOU ARE LOST IN A MAZE!","Y
OU ARE LOST IN A MAZE!"
9016 DATA"YOU ARE SPLASHING ABOUT IN
A COLD, RUSHING STREAM!
NOTHING YOU DO SEEMS TO STOP YOUR PER
ILOUS RIDE TOWARDS A
NEARBY STONY CAVE ENTRANCE!","RUSHING
 STREAM"
9017 DATA"YOU LIE ON THE SANDS OF A D
ARK, SLIMY CAVERN BY A
STREAM. THE WALLS ARE COVERED WITH DI
SGUSTING OOZE. THERE IS
A HOLE IN THE NORTHERN ROCKS AND A PA
TH NORTHEASTWARD.","SLIMY CAVERN"
9018 DATA"SWEAT BEADS ON YOUR FACE AS
 YOU STAND IN A STEAMY
CAVE. SMOKE RISES FROM A HOLE IN THE
FLOOR, AND THERE IS
ANOTHER RAGGED HOLE IN THE ROOF WITHI
N REACH. THERES ALSO
A PATH GOING SOUTHWEST.","STEAMY CAVE
"
9019 DATA"YOU ARE CROUCHING ON A FIER
Y SPIRE, A PINNACLE
SURROUNDED BY FLAMES! A LOW ROOF WITH
 A HOLE HANGS A FOOT
ABOVE YOUR HEAD. IT IS UNBEARABLY HOT
!","FIERY SPIRE"
```

Chapter 10



Improving the Program

Computer programmers are the most dissatisified class of people you are likely to meet. It's not enough for them to have a program that plunges the player into a carefully constructed alternate reality for hours. That's peanuts! The real challenge for programmers is to write the program better, faster, more efficiently, with more style.

Now, I'd be foolish to claim that Basements and Beasties is as optimized and efficient as can be. In fact, there are a number of tricks that can be employed to squeeze the program even further. In this last chapter, I really push BASIC to its limits and find out just how much complexity I can get at high speed with little memory.

Of course, I ought to warn you where all of this is leading. BASIC has served us well through these past chapters, but in my heart of hearts, I know that Basements and Beasties ought to be enlarged and rewritten in assembly language. Some remarks toward the end of this chapter address how you might begin to attempt this task.

One preliminary comment is in order. Several improvements and modifications are described in this chapter, along with BASIC code to implement them. Not all of them are compatible simply by adding these new lines of code. Before attempting to implement everything at once, review what variables have been changed and what other sections of code are affected. One small change creates a ripple-effect that could leave your adventure program adrift in a sea of syntax errors!

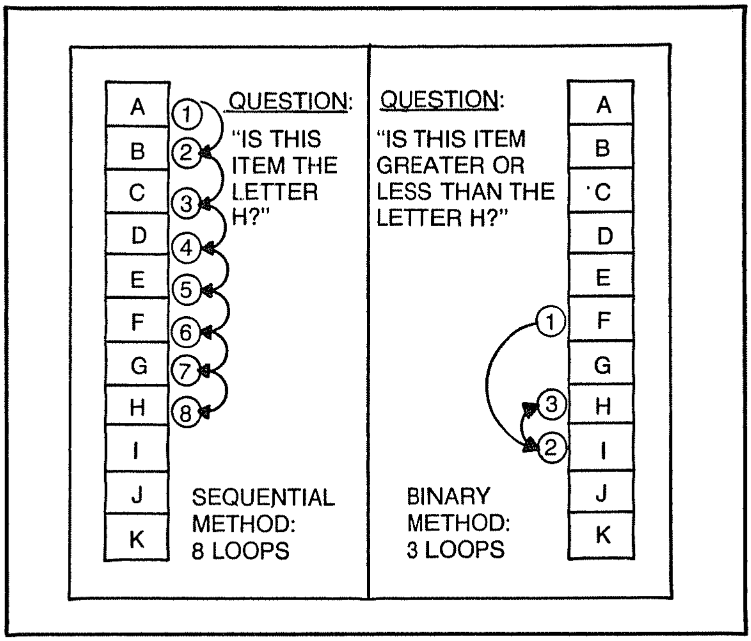
A FASTER WORD SEARCH

In Chapter 3 you looked at the command subsection of Executive. When a one- or two-word command is input, that section of code performs two initial functions. First, it divides the input into its separate words (if that is necessary). Second, it takes the first word and tries to identify it.

The key to the program's interpretation of inputs is the data block known as the word table. All words that the program is to understand are listed in this table, paired with an ID number that aids in definition. If a word is not found in the word table, the program usually responds in ignorance with a message like, "WHAT DID YOU SAY?"

There is a subroutine, called Idword, that is used to access and use the word table. Given a word in the variable A$, Idword searches the entire table, attempting to find a word that matches A$. If it succeeds, it sets the variable *N* to the accompanying ID number. If it fails, N is set to zero.

What is the fastest way to search a table? One method is a simple *sequential search*. Every word in the table is checked, from



A
B
C
D
E
F
G
H
I
J
K
QUESTION:
"IS THIS ITEM THE LETTER H?"
SEQUENTIAL METHOD: 8 LOOPS
QUESTION:
"IS THIS ITEM GREATER OR LESS THAN THE LETTER H?"
BINARY METHOD: 3 LOOPS

Fig. 10-1. Sequential versus binary searching. (Note that binary searching seeks the approximate halfway point of each table segment it checks, rounding up if necessary.)

```
1080 IFLEN(A$)>5THENA$=LEFT$(A$,5)
1082 A=2:B=1:GOSUB1040
1084 READB$,N:IFB$="."ORB$=A$THENRETU
RNELSEIFB$>A$THENN=0:RETURN:ELSE1084
```

Fig. 10-2. The subroutine Idword revised. Note that the word table must be alphabetized for this change to function.

the very start of the table, until a match is found. If no match occurs the entire table is read, which is a waste of time.

A second method is called a *binary search*. In such a case, the words in the table are set in alphabetical order. The middle entry is checked for a match first. Depending on whether the sought word is alphabetically less or greater than that entry, one half of the table is eliminated from consideration. The middle entry of the remaining half is checked, and so on, until a match does or does not occur. The binary search method is very fast, indeed. (Fig. 10-1 compares these two search methods.)

Now, after that big build-up, this chapter does *not* give the code for a binary search of the word table. Why not? The primary reason is that the items in the table are all in a series of DATA lines. Even with improved data access methods used to POKE values into the BASIC data pointer, it is cumbersome to find the middle entry of the table, and the middle entry of half of the table, and so on. The READ statement itself is by nature sequential. The code it would take to maneuver the data pointer into a binary search is too complex to justify it as the method.

So what do I suggest? As usual, a good compromise is better than no progress at all. Take a look at Fig. 10-2. This is the code for an improved version of the subroutine Idword. The underlined statements are new and the rest is unchanged. Apart from these additions, one other change is necessary: all words in the word table must be placed in alphabetical order. In this way, the new Idword can tell whether it has looked too long and too far for the desired word, and when to give up.

Line 1084 is the crucial segment. Note that the search begins at the very top of the table, just like a regular sequential search. For each iteration of the search, a word from the table is read into B$ and its accompanying ID number is read into N. The very last word in the table is a period with an ID number of zero. Thus, with each iteration, Idword checks for one of two conditions. If a match occurs between B$ and A$, the subroutine returns, and N equals the ID number. Also, if B$ is a period, the search is over and *N* equals zero to indicate failure. All of this is just as it was before.

Here's the improvement. If neither of the above cases occur, then maybe the word is still somewhere in the rest of the table. But, you can eliminate some searching if you compare the word's alphabetical position to that of the table entry. If the sought word begins with a "D" and the previous table entry began with an "E", you know (assuming an ordered table) that further searching is unnecessary. The search-time savings average out to about 50 percent.

How can this work? Looking at the listing, you see the expression, IF B$>A$. In Microsoft BASIC the comparison operators ">" and "<" can be used to compare two strings for alphabetical relationship. Thus, the word "DROP" is alphabetically "less than" the word "LOOK." In cases of shortened words, the dictionary order applies: "ACT" comes before "ACTION" and is therefore alphabetically less.

So, the new Idword makes a final test. If the table entry just read is alphabetically greater than the desired word, the word cannot be in the remainder of the table. The variable *N* is set to 0 to indicate search failure and Idword returns. If you think about it, this is similar to the way in which you verify that a word is not in a dictionary. Once you are in the general area where the word should be, you look for a match. If you find that one word and the next is greater, you don't need to search anymore. (Imagine if dictionaries weren't ordered alphabetically!)

This upgrade to Idword is one that can be made to the program immediately. Once the word table is reordered alphabetically (get to work), this improvement works without problems. Remember that new words added to the table must be placed in the proper position.

HEAVY OBJECTS

In Chapter 6, which deals with how the adventurer can affect the scenario, a lengthy explanation is provided for the handler called Take. This handler, if you recall, is invoked by the keywords TAKE or STEAL, and it controls the players ability to pick up objects and tote them in his carry-sack. A number of limitations are placed on the player in this regard. For instance, the adventurer is forbidden to carry creatures.

The major parameter that limits the act of carrying, however, is the maximum amount of five objects. The variable CT(2) is carefully updated each time an object is taken or dropped. The handler Take does not permit a new item to be picked up if CT(2) is already at the maximum total of five.

This sort of limitation is a simple one to maintain, but it lacks realism. After all, the objects in Basements and Beasties range from small rings to heavy golden cubes to unwieldy magic axes. It would certainly be more realistic to limit the player by the weight and bulkiness of an object rather than a simple total.

How might this be done? For one thing, each object must be assigned a number that approximates its value in terms of portability. As a player gathers objects, their numbers (which may be called mass numbers) are totaled and recorded. If the addition of a new object with its unique mass number would cause the total to exceed some arbitrary limit, a message warns the player that he cannot pick it up without collapsing altogether!

Figure 10-3 shows the mass chart, in which each of the portable objects in the scenario is assigned a mass number. These approximate mass numbers range from 1 to 50 and are an evaluation both of weight and ease of carrying. Obviously, the numbers are arbitrary. The program descriptions never tell, for instance, how big the Enchanted Grenade is or of what it is made. You may wish to revise these numbers altogether.

That's the simple part. Where do you put these numbers so that the handler Take (when it is modified) can access them? Fortunately, you planned ahead. The object status array consists of OB(X,0) and OB(X,1) for all objects. OB(X,1) gives the object's location, but

	OBJECT	MASS		OBJECT	MASS
1	CROWN	25	7	COIN	5
2	CUBE	40	8	HOUR-GLASS	20
3	BEETLE	15	9	TORCH	15
4	BELT	20	10	AXE	20
5	RING	10	11	KEY	5
6	ONYX	10	12	GRENADE	25

Fig. 10-3. Proposed mass chart assigning an arbitrary mass factor to each portable object.

```
3000 DATA 4,25,7,40,20,15,11,20,5,10,
19,10,7,5,6,20,2,15,3,20,10,5,12,25,4
,0,18,0,14,0,6,0
```

Fig. 10-4. The revised object initialization block, such that the first 12 objects are assigned mass numbers.

OB(X,0) is unassigned. The mass number for each object can be stored in these unused elements of the object status array.

How do you put the mass numbers into the proper variables? In the initialization code of Basements and Beasties the object status array is filled from a data block, the object initialization block on line 3000. Up until now, every other element in that data block was a zero, unused; the other data are the beginning room locations for all objects. Now replace those zeroes with the appropriate mass numbers.

Figure 10-4 shows the new version of line 3000. The first data is the beginning location for object 1; the Crown of Jewels begins at room 4. It has a mass number of 25. Once this block of data is read, OB(X,0) can quickly be checked to determine if the object in question can be carried.

The handlers Take and Drop look a bit different. It goes without saying that the old system using CT(2) for the total number of objects carried is eliminated. Thus, any references in other routines that increment, decrement, or set CT(2) need to be removed.

Figure 10-5 shows the changes that need to be made in Take and Drop. Let's look at Take first, since it is modified the most. The handler Take must determine if the addition of this new object is too much for the player to handle. You must make two assumptions. First, assume that the mass numbers are totaled and stored in CT(2) each time an object is taken. Second, assume an arbitrary carry-sack maximum of 75 total mass points.

In the original Take, it did not matter what object the player tried to lift: if it pushed his total above five objects, it was prohibited. The old Take did not need to decode what the object was until this case was dismissed. The new Take, however, must know what the object is before it decides if it is too much to carry. Line 240 begins by a decoding of word 2 stored in TX\$(3). The subroutine Idword is used to locate the object's name in the word table and return with its ID number in variable N. For objects, the ID number equals the object number.

With this information, Take can now perform a comparison. CT(2) contains the player's total burden in terms of mass points. If the addition of this new object's mass, as found in OB(N,0), results in a total that exceeds 75, it is too much to bear. If so, Take proceeds to display message 36, as it did before: "YOUR ARMS ARE FULL . . . YOU CAN CARRY NO MORE." On the other hand, if the limit of 75 mass points is not exceeded, the handler goes ahead and allows the object to be lifted. To keep track of the load, CT(2) is increased by the added mass in OB(N,0).

Next, look at the handler Drop. If all of the prerequisites are met, Drop permits the specified object to be taken away from the carry-sack. Again, CT(2) is adjusted to keep track of the total burden. The mass value of the object as stored in OB(X,0) is subtracted from CT(2).

There are other routines that are affected by this improvement, notably the Resur handler that resurrects the player with an emptied carry-sack. Any other such modification is simple to make, following the example of Take and Drop.

RUN-TIME BASEMENTS

The small computer market does not lack for adventure programs of every size, variety, and degree of complexity. If you study the available programs, though, you'll discover that they generally fall into one of two categories. The first category consists of *fixed labyrinth* games, like Basements and Beasties, programs in which

```
240 A$=TX$(3):GOSUB1080:IFN>9999THENB
=7:GOTO242:ELSEIFCT(2)+OB(N,0)>75THEN
B=36:GOSUB1100:GOTO104:ELSEIFN>12ANDN
<17ORN=18THENB=40:GOTO242
241 IFN=17THENB=8:GOTO242:ELSEIFOB(N,
1)=21THENB=9:GOTO242:ELSEIFOB(N,1)<>C
T(0)ORN=0THENB=12:GOTO242:ELSEOB(N,1)
=21:B=11:CT(2)=CT(2)+OB(N,0)
242 GOSUB1100:GOTO104

260 A$=TX$(3):GOSUB1080:IFN>9999THENB
=7:GOSUB262:ELSEIFOB(N,1)<>21THENB=10
:GOTO262:ELSEIFN=12THEN540:ELSEOB(N,1
)=CT(0):B=11:CT(2)=CT(2)-OB(N,0)
262 GOSUB1100:GOTO104
```

Fig. 10-5. The revised versions of the handlers Take and Drop, respectively, allowing for mass number assignments to objects.

the treasures, creatures, and pathways are the same every time they are played. As such, programs of this first type are like puzzles that, once solved, are replayed primarily to increase speed and score.

The second category, though, consists of *variable labyrinth* games, in which the treasures and situations change locations each time the program is run. Since the layout of the scenario is determined randomly when the game is first run, these programs are said to have *run-time scenarios.*

There are a few drawbacks to run-time adventures. First, they are more battle-oriented than solution-oriented. That is, specific tests of the adventurer's cleverness are not often included, since these usually imply a fixed room location with fixed entrances and exits. (Consider the narrow ledge in Basements and Beasties.) Second, they sometimes downplay the role of on-screen description of the rooms, since such descriptions cannot be explicit about doorways, which are always changing.

These objections are not universal, however, and you may be wondering if Basements and Beasties might not be a bit more interesting if there were more random factors included. For you, let's discuss some ways of creating run-time basements.

At least in its present form, your program does not yield to random pathway designation. The room descriptions tell where every doorway is. These references all have to be deleted. Plus, the function of obstacles depends heavily on the directions from which the player exits a room.

For these reasons, it is best rather to think in terms of random placement of objects, and specifically the treasures, at various places throughout the scenario. Each time the game is played the adventurer does not know where the treasures are, and the chances of him recovering them easily vary each time.

In Fig. 10-6, you can see a line of code that can be placed in the initialization section of the program. A loop is set to affect objects 1 through 8, which are the treasures. For each treasure, its location, as found in variable OB(I,1), is set randomly. The expression RND(17)+3 provides a room number from 3 to 20. This prevents the player from finding treasures without even going down into the basement; rooms 1 and 2 are excluded. (Of course, if you don't care, you can substitute the expression RND(19)+1, which provides a room number of from 1 to 20. In fact, the expression RND(20) permits some treasures to be placed in limbo, room 0, so that in some rounds of the game the total possible score is lessened.

```
12 FORI=1TO8:OB(I,1)=RND(17)+3:NEXTI
```

Fig. 10-6. Addition to the Initialization code to randomize object location, specifically the treasures.

There is a way to make the run-time basement even riskier. For the FOR-NEXT loop in the example, substitute a range of from 1 to 12. This provides for the random placement of such necessary items as the Key and the Axe. Some miserable possibilities can turn up in this case. For instance, the Key may be placed in a locked room! Also, the player may have to do some hard searching just to find the Axe to defend himself. However, do the poor adventurer a favor: add the statement "CT(9,1)=2" after the loop. This ensures that the torch is always available above ground. You don't expect the pitiful hero to grope around for it in the dark, do you?

Aside from this sort of random assignment of object locations, there is one more sort of run-time approach that can be incorporated into Basements and Beasties. Several programs have been written that store entire scenarios in data files on tape. The player loads a main program that in turn loads in whichever scenario the player may choose. The net effect is that of a multi-floor scenario that is limited only by the number of available files on tape.

How can this concept be implemented? Basically, all scenario information, including room and object descriptions, travel table, obstacle list, and so on, can be created on tape by using a special program. The word table and message block are a part of the main program that stays in memory from the start, but extensions to both of these must be loaded from tape to support specifics of each floor of the basement.

The key to the transition is that almost everything that now resides in DATA statements will reside only on tape, until loaded. At that point, the data is stored in numeric and string variables. Thus, a whole set of arrays needs to be set up to receive data from tape. For example, for the 20 rooms (per floor) there needs to be an array RD$(20,1) for room descriptions. RD$(x,0) contains the long form, RD$(x,1) the short form. A similar array OD$(16,1) serves for object descriptions, and so on.

A CONDENSED TRAVEL TABLE

If your goal is to find ways of reducing the use of memory usage, there are plenty of tricks that can be applied. For instance, thanks to the inclusion of the PEEK and POKE statements in BASIC for the

TRS-80, block of memory can be accessed and altered without the use of other BASIC structures, such as, arrays and DATA lines. In several cases, data managed by PEEK and POKE can be set up more efficiently than by other means.

Let's take a look at the best example of this: the travel table. In its present form, the travel table consists of 20 DATA lines, one for each room in the scenario. Each line contains eleven items, and each item is a number from 0 to 23. Now, if you add together the number of bytes taken for each number, plus the memory needed for DATA block overhead, you discover that the travel table fills about 595 bytes of memory.

A lot of this space is really wasted. Since each data item is less than 256, each needs only 1 byte. If so, the total memory consumption would only be about 220 bytes: 11 items multiplied by 20 rooms. The rest of the space is taken up by structures to permit the items to be read by the READ command as DATA elements. Think of it: 375 bytes are consumed in things like BASIC line numbers and pointers, DATA keywords, and commas to separate the items! If PEEK and POKE are used, most of this is superfluous, and a savings of well over 60 percent could be realized.

Where do you put this magical block of bytes, this data block without DATA structures? It may surprise you to hear it, but you can put the data bytes right into a BASIC line. As long as you know the exact memory location that begins the BASIC line, you can PEEK the contents of the line to your heart's content.

This unusual assertion needs some bolstering. The plain fact is, BASIC for the TRS-80 really only makes a few stipulations concerning what can be put into a line. The first restriction is a length limitation of 255 characters (or bytes) maximum. You only need 220 of these for the travel table, so that is no problem. Second, any character at all can be placed into a BASIC line (even special control characters with ASCII values below 32) with one notable exception—character zero. This is because BASIC uses a zero byte to determine the end of the line. As long as you don't POKE zero values into a line, you are free to POKE anything else from 1 to 255.

The third stipulation is that the contents of a BASIC line can be nonsensical as far as BASIC syntax, as long as the program does not try to execute the line. That is, you can POKE numbers into a BASIC line that spell out gibberish, and it does not crash the program—as long as that line is avoided in execution. So you can fill a BASIC line with bytes that make sense to a data-access routine of some sort, without worrying that it might somehow confuse BASIC.

You don't even need a REM statement to protect the line, if you simply stay away from it.

One hasty interjection should be made. These stipulations refer strictly to the proper running of a program. I haven't said anything about listing the program. Obviously, if I POKE some unusual bytes into a BASIC line, the listing may appear with graphics blocks or be unreadable altogether due to control characters like clear-screen. A garbled listing does not prevent the program from running, though. In fact, those strange lines can even be edited from BASIC without trouble.

Now, let's get specific. In place of 20 DATA lines from 5000 to 5038, I propose one BASIC line, numbered 5000. Its exact location in memory is stored in the data-access array DA(x) anyway, thanks to the initialization segment of the program. The line will contain 220 bytes of information, corresponding to the present 220 DATA items in the travel table.

Wait a minute! I hear some objections from somewhere. First, some of those values in the travel table are zeroes, and that's not permitted as a byte in a BASIC line. Second, how are you to type all of this in as you write the program? There are no keys for typing in all of the characters with ASCII values less than 32. What do you do?

There is one answer to both questions: encode the data a bit. You already know that the numbers in the travel table range from 0 to 23. If you add 65 to every value, the range is from 65 to 88—and characters 65 to 88 are the letters A through X. Those letters are safe on a BASIC line and are easily entered from the keyboard. Whenever you access the travel table bytes using PEEK, though, you need to remember to subtract 65 to return to the original value.

So much for theory; now on to code. Figure 10-7 gives two sections of Basements and Beasties. The first is the actual travel table, encoded and stored on one BASIC line, number 5000. Using the correspondence factor that the letter *A* represents a zero, the letter *B* a one, and so on, compare the listing to the DATA items in the present version of the program.

The second listing is the new form of the subroutine called Travec. Recall that when the program presently needs to access the travel table, it sets variable *D* to a number from 1 to 11, to choose the specific item from a table row. (The row itself corresponds to the room the player is in at the moment, which is stored in CT(0).) Then Travec is called, which finds the proper DATA block and the proper row, fetches the item, and stores it in variable *A*.

In the new version of Travec, the inputs and outputs are the

```
5000BCCBBBBBADJCCCCCCBBCAIJAAEKAAAABAI
    AFAALADAAAEAAAAAEAAAAFAAAMAAAAAAXD
    AAAOAAAAAAADAAAAOAAACAIJAQPJAAJAAH
    XXXQRRRDAREEAAAAAAAAAAAAANASAAGAAH
    AAAAAAMAAAGIAAATAAHATEPAPAPQJAAAA
    PQQAQAKJAABSSSSSSSSSSAMTAAAAAAAAA
    OAAAASAAOUJWWWWWWWWTWI

 1120 A=(CT(0)-1)*11+(D-1)+DA(1)+4:A=P
 EEK(A)-65:RETURN
```

Fig. 10-7. New encoded travel table and the revised version of the subroutine Travec. Line 5000 should be typed with no spaces at all.

same as before, but the method has been changed. The new Travec must calculate the memory location at which to find the specific travel vector byte. The memory address for the beginning of line 5000 is already stored in DA(1), but a factor must be added to this address to locate a particular byte.

Take line 1120 expression by expression. First, although nothing actually divides the bytes in the new travel table, they are still organized in series of eleven, one series of eleven per room. If the bytes for room 1 start at the beginning of the line, the bytes for room 2 start eleven bytes later, and so on. The expression (CT(0)−1)*11 helps Travec skip to the exact 11-byte series that matches the present room. If it is room 1, the expression equals zero, meaning that the bytes for room 1 are right at the start of the line with no addition needed. For room 2, the expression equals 11, meaning that 11 must be added to the memory address of the beginning of the block to get to the series for room 2.

Once the right 11-byte series is found, the expression D−1 is added. Remember that D is a number from 1 to 11. This expression converts it into the form 0 to 10 to be added to the beginning of the series. For example, if you want travel vector 1, it is the very first byte in the series for the present room, and no addition need be made or skipping done.

Finally, the expression DA(1)+4 adds all of the preceding to the exact memory address of the first of the 220 bytes. DA(1) contains the memory address of a part of the BASIC line called the *line vector* (see elsewhere in this book for a fuller description), and the actual contents of the line do not start until 4 bytes later in memory. All of the expressions added together provide the address of a specific character; this address is temporarily stored in variable *A*.

Reading the memory location is simple; the statement PEEK(A) accesses the location and provides the value of the character stored there. The value is from 65 to 88, and you need it in the form 0 to 23. So, 65 is subtracted from the PEEK value, and the result is stored in variable *A*. Travec is finished and returns to the calling program.

The user can expect two advantages to this approach. First, you saw the vast savings in memory; some 370 bytes worth is nothing to sniff at. Second, the user will probably notice a speed difference in the execution of motion commands. Before, Travec had to call another subroutine to move the data pointer down to a specific DATA line, then READ across to the item using a FOR-NEXT loop. Now, Travec simply evaluates an expression of medium complexity, uses the value to read a byte, and subtracts a fixed value from the result—which takes a lot less time than some of the FOR-NEXT loops of the old method.

By the way, the listing of line 1120 is not even as simplified as it can be; I left it that way for ease of explanation. The first part may also read A=(CT(0)-1)*11+D+DA(1)+3.

USR RUSES

For the entire book, I have been dealing with ways to accentuate BASIC, because BASIC is so slow. I have seen adventure programs that take an average of 20 seconds to respond to any one command. This makes a boring game: hence the search for streamlining devices.

In the final analysis, of course, machine-language routines are far faster than interpreted BASIC, and the best adventure program is one written entirely in assembly language. Not everyone is ready to tackle that sort of task, though, and BASIC makes things easier.

Fortunately, Microsoft BASIC provides for a third alternative, *hybrid programming*, which allows BASIC to call fast machine code routines on occasion. The statement that supports this facility is USR(N). Using this statement, BASIC can relinquish control of the processor to assembly routines designed to handle more complex oft-used functions in the most efficient manner.

You probably know some things about using USR(N). You know that you need to POKE the starting address of the routine first in two-byte form into memory locations 16526 and 16527. You know that the expression X=USR(N) can be used, and that the variables *X* and *N* in that expression may be affected by the called routine.

One frequently discussed aspect of USR(N) is the question, "Where do I put the routine?" Machine-coded routines can be stored in upper memory, but you must protect that space with the MEMORY SIZE option, or else BASIC steals the memory for string space. You can POKE routines into a string variable, as long as that variable is left alone. There is waste involved, because the program must include a BASIC subroutine to READ the machine codes from a DATA line and to POKE them into the string. In the end the routine is stored in two places: in a form able to be executed in the string and in the DATA line as now-useless numbers.

Now for a rare piece of USR news. (It should come as no surprise to you after reading the previous section of this chapter.) A machine-code subroutine can exist right in a BASIC line! That way, the routine is ready to use the minute the program is loaded from tape; it does not have to be constructed by POKE commands and FOR-NEXT loops.

As I've shown once before, there are restrictions on this sort of thing. Your machine-code routine cannot exceed 255 bytes, or else it will not fit in a single BASIC line. It must be avoided by BASIC, or else a syntax error occurs; the line must either be skipped, or it must be protected by a beginning REM marker. Finally, and most importantly, no zero bytes are tolerated. An ill-placed zero byte confuses BASIC utterly. It takes some care to write an assembly routine that avoids the use of zero byte, but it can be done.

The use of USR(N) requires that the address in memory of the beginning of the routine be known. To simplify things a bit, it is a good idea to put machine-language routines into the first lines of a program. You know that BASIC storage starts at 17384 (or, for Model I users, 17128), so specific routine addresses can be calculated from this fact. Since the program tries to execute these lines if you type, "RUN," you need to protect them by starting each line with the REM statement. With four bytes taken for the encoded line number and line vector, plus two bytes for the REM marker, a good place to start the routine is at location 17391 in memory.

The next question is, how do you get the routine into the line? The answer is, with a temporary POKE loop. Look ahead, briefly, to Fig. 10-8. Use this short block of code to POKE a routine into the first BASIC line in a storage, assuming that the line is created already and filled with enough spaces to hold the routine. The last thing that the code does is delete itself, because it is no longer needed. The resulting BASIC line can be saved or loaded from tape with no ill-effects, other than the rather distorted effect it produces during a LIST command.

```
20 RESTORE: FOR I=17391 TO 17308: REA
D N: POKE I,N: NEXT: DELETE 20-26: EN
D
22 DATA 42,251,64,35,126,35,70,35,254
,65,32,5,120,254,68,40,7,94,35,86,35,
25,24,235,30,9,175,87,25,94,35,86,27,
213,42,249,64,126,95,35,70,35,78,35,2
54,3,32,9,120,183,32,5,121,254,65,40,
5,175,87,25,24
24 DATA 231,70,35,94,35,86,225,213,12
0,254,6,56,2,6,5,72,65,209,213,126,25
4,44,40,8,183,32,8,35,35,35,35,35,35,
24,237,254,46,32,2,225,201,26,19,190,
35,40,12,43,35,126,254,44,40,221,183,
40,218,24,245,16
26 DATA 214,126,254,44,40,8,183,32,23
5,35,35,35,35,35,35,205,90,30,225,213
,42,249,64,126,95,35,70,35,78,35,254,
2,32,9,120,183,32,5,121,254,78,40,5,1
75,87,25,24,231,209,115,35,114,201

10 CT(0)=1:CT(12)=RND(10)+10:CLS:POKE
16526,239:POKE 16527,67

1080 N=0:N=USR(0):RETURN
```

Fig. 10-8. This POKE routine creates a machine-code subroutine in line 1. Line 10 prepares BASIC for the USR statement, and line 1080 issues the call to the new subroutine.

The procedure is simple. The programmer creates line 1 of the BASIC program as a REM line full of following spaces. The number of spaces depends on the number of bytes needed by the machine-code routine. Then, he types in the lines in Fig. 10-8, making certain that these DATA lines are the earliest DATA lines in the whole program. When he types, "RUN 20," the spaces in line 1 are replaced by machine-code bytes, and finally the POKE code self-deletes. Line 1 is ready for access by a USR call.

LOOKING UP WORD QUICKLY

If you *do* want to speed BASIC up by sprinkling in a few machine-language segments, what functions should you augment? BASIC is plenty fast in most cases, but what you want is to shorten the visible delay between command and response that is so obvious in BASIC adventures. How do you do that?

If you trace the execution of Basements and Beasties by using the TRON function of Microsoft BASIC, you'd find one routine in particular that seems to take forever to execute. That routine is Idword, the subroutine responsible for matching an input word with a word table entry. If you have not alphabetized the word table, this routine can take three or four seconds maximum, just to determine if the word is in its vocabulary. Moreover, the maximum time occurs every time the input word is not recognized. The player can enter "STUPID GAME" and wait several boring seconds before getting the response, "WHAT DID YOU SAY?" What you need is a machine-language version of Idword. Such a routine can reduce the word table scan time to mere milliseconds.

Figure 10-8 provides a BASIC routine to POKE a machine-language version of Idword into memory. Remember from earlier discussion that a substitute REM line full of spaces must be prepared to receive the information. The machine code in this case requires 174 bytes, starting at memory location 17391, which is shortly after the actual REM indicator in memory. Be sure to create line 1 with this many spaces, plus one for the REM, at least.

When the BASIC routine is RUN, line 1 is filled with the new subroutine, and the BASIC lines self-delete. The strange bytes in line 1 do not interfere with the saving, loading, or running of the program in any way. Just remember that the command LIST results in garbage for line 1, but everything else should LIST fine.

Figure 10-8 also gives the few changes that need to be made in the program as a whole to accommodate the new subroutine. The initialization section of Basements and Beasties must POKE the proper values into the USR pointer, so that a USR call results in a call to memory location 17391. Secondly, the present BASIC version of Idword must be replaced by the simplified line as shown. Note that the variable *N* is set equal to zero. As is explained in a moment, this assures that the variable *N* exists in memory for the new subroutine to find and manipulate. That way the routine need not be able to create new variables, a task that takes some complex operations.

(Some of the values in Fig. 10-8 are different for users of the TRS-80 Model I. This is because the Model I begins its BASIC storage area about 256 bytes earlier than the Model III. Therefore, the POKE loop that stores the machine-language routine in memory must begin with memory location 17135, not 17391. Also, when setting up the USR pointer, you must POKE the values 239 and

```
VARIAS   EQU    16633
ARRAYS   EQU    16635
         ORG    17135
IDWORD   LD     HL,(ARRAYS)
ID1      INC    HL
         LD     A,(HL)
         INC    HL
         LD     B,(HL)
         INC    HL
         CP     65
         JR     NZ,ID2
         LD     A,B
         CP     68
         JR     Z,ID3
ID2      LD     E,(HL)
         INC    HL
         LD     D,(HL)
         INC    HL
         ADD    HL,DE
         JR     ID1
ID3      LD     E,9
         XOR    A
         LD     D,A
         ADD    HL,DE
         LD     E,(HL)
         INC    HL
         LD     D,(HL)
         DEC    DE
         PUSH   DE
         LD     HL,(VARIAS)
ID4      LD     A,(HL)
         LD     E,A
         INC    HL
         LD     B,(HL)
         INC    HL
         LD     C,(HL)
         INC    HL
         CP     3
         JR     NZ,ID5
         LD     A,B
         OR     A
         JR     NZ,ID5
         LD     A,C
         CP     65
         JR     Z,ID6
ID5      XOR    A
         LD     D,A
         ADD    HL,DE
         JR     ID4
ID6      LD     B,(HL)
         INC    HL
         LD     E,(HL)
         INC    HL
         LD     D,(HL)
```

```
         POP    HL
         PUSH   DE
         LD     A,B
         CP     6
         JR     C,ID7
         LD     B,5
ID7      LD     C,B
ID8      LD     B,C
         POP    DE
         PUSH   DE
ID9      LD     A,(HL)
         CP     44
         JR     Z,ID10
         OR     A
         JR     NZ,ID11
         INC    HL
         INC    HL
         INC    HL
         INC    HL
         INC    HL
ID10     INC    HL
         JR     ID8
ID11     CP     46
         JR     NZ,ID11A
         POP    HL
         RET
ID11A    LD     A,(DE)
         INC    DE
         CP     (HL)
         INC    HL
         JR     Z,ID13
         DEC    HL
ID12     INC    HL
         LD     A,(HL)
         CP     44
         JR     Z,ID9
         OR     A
         JR     Z,ID9
         JR     ID12
ID13     DJNZ   ID9
         LD     A,(HL)
         CP     44
         JR     Z,ID14
         OR     A
         JR     NZ.ID12
         INC    HL
         INC    HL
         INC    HL
         INC    HL
         INC    HL
ID14     INC    HL
         CALL   1E5AH
         POP    HL
         PUSH   DE
```

Fig. 10-9. The source listing of the new machine-language Idword.

```
        LD      HL,(VARIAS)             LD      A,C
ID15    LD      A,(HL)                  CP      78
        LD      E,A                     JR      Z,ID17
        INC     HL              ID16    XOR     A
        LD      B,(HL)                  LD      D,A
        INC     HL                      ADD     HL,DE
        LD      C,(HL)                  JR      ID15
        INC     HL              ID17    POP     DE
        CP      2                       LD      (HL),E
        JR      NZ,ID16                 INC     HL
        LD      A,B                     LD      (HL),D
        OR      A                       RET
        JR      NZ,ID16                 END
```

66—not 239 and 67. The code for the routine itself does not change; it is relocatable, or address-independent.)

The BASIC user who has never dabbled in machine language can use the routine without being concerned with how it works. For the bold, however, I think it's only fair to include a listing of the assembly code that produced the routine, along with a brief description, which is given in Fig. 10-9.

Consider briefly the requirements of the subroutine, which I continue to call Idword. The calling program stores the word to be searched for in the string variable A$. Idword compares this word to each of the entries in the word table, until it either finds a match or reaches the end of the table. If the word is found, the accompanying ID number is read and stored in the variable *N*; a search failure sets *N* to 0.

A machine-language table-search is fairly easy to contrive. The task that takes some thought, though, is how to interface to those variables! Where are they in memory? What is their format? How do you find them and change their values?

For the purposes of the discussion, refer to the diagrams in Fig. 10-10. You need to work with two structures in memory: straight variables, such as *N* and A$, and variable arrays, such as DA(n). These structures are stored by BASIC in the free memory space that follows the actual program lines. Simple variables are stored first, and not in any particular order; arrays come next. BASIC maintains two pointers to help locate the memory areas where these structures are. Locations 16633 and 16634 contain the address where simple variables start. I call this pointer Varias. Locations 16635 and 16636 point to the start of the arrays, and this pointer is termed Arrays.

Note, by the way, that strings do not actually reside in the immediate area where other variables are. Rather, the entry in that area provides an address where the string can be found. Strings are stored up near the top of available memory.

The first thing that the new Idword must do is find the beginning of the word table. This address is stored in array variable DA(2), since the word table is DATA block 2. Idword loads the beginning of array space from the pointer Arrays. Then it searches for an array entry that has the letter A in the second byte and the letter D in the third byte, the reversed form of the name DA. The ASCII codes 65 and 68 correspond to these letters. Each array that does not correspond to this name is skipped; this is done by using the value stored in the fourth and fifth bytes, which tells how many bytes are left in that variable. Idword loops until it finds DA(n).

When it has found the array, it needs to locate the value of the specific entry DA(2). The actual values of the array elements begin five bytes after the name characters, beginning with the value of DA(0). Since each value takes two bytes, the value of DA(2) is found a total of nine bytes after the name characters. So Idword skips this far ahead, reads the two-byte value, and stores it on the stack using the PUSH instruction.

Now you need the location in memory where A$ begins so you can do comparisons. Using the pointer Varias, the routine performs another search looking for two factors: a first byte equal to three, indicating a string variable, and second and third bytes equal to zero and 65 respectively, indicating a name character of A. (Single-letter variables fill the remaining byte with 0). If a match of this kind does not occur, the variable is skipped. This is done by skipping ahead by the same number of bytes as the value of the first byte, the type identifier. Type 2 variables, or integers, use two bytes to store a variable; type 3 variables or strings use three bytes to point to the string. Thus, the identifier number can tell Idword how far to advance in order to find the next variable. (Since the BASIC program contains a DEFINT statement, expect to see these two types of variables only, not single or double-precision variables.)

Once the variable A$ is found, the value of the fourth byte is stored; this value tells how long A$ is. The next two bytes, which give the starting address of the string in memory, are stored. Finally, the length of A$ as stored in register B is changed; so it is no larger than five. In this way only the first five letters of any input word are considered significant. To save space no word-table entry

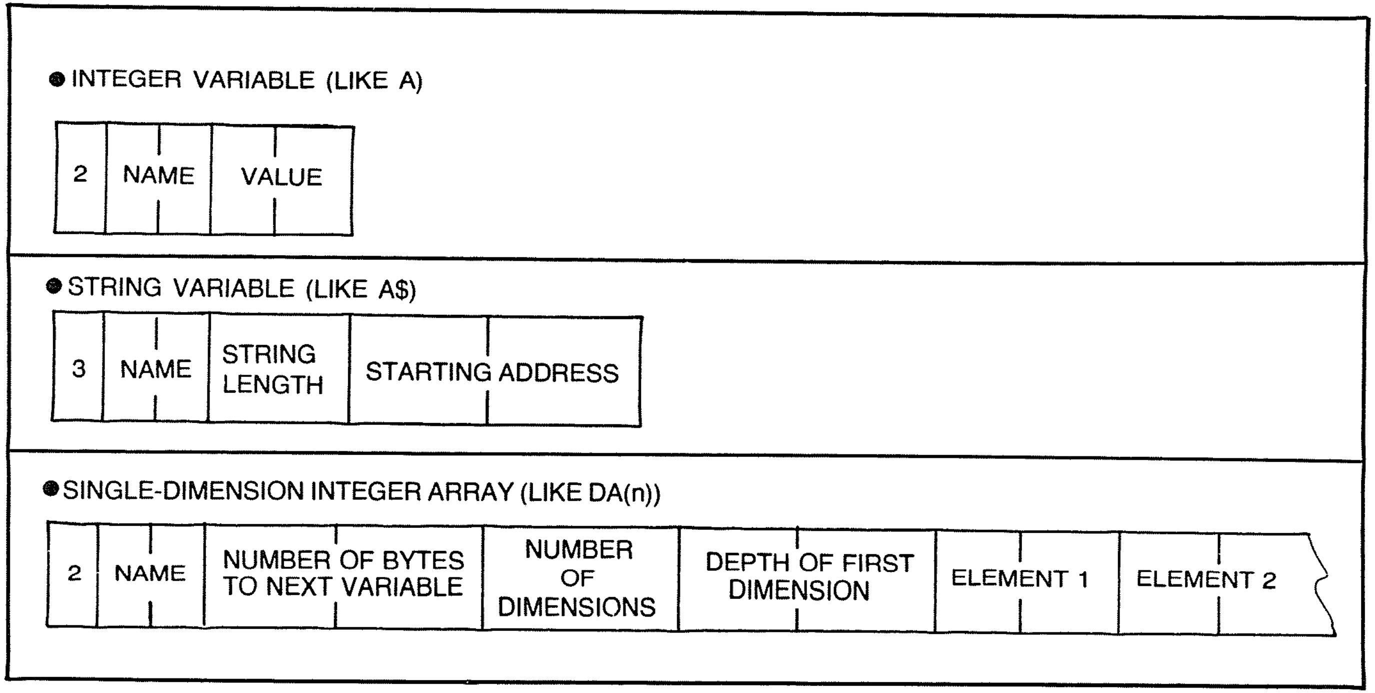

INTEGER VARIABLE (LIKE A)
2
NAME
VALUE
STRING VARIABLE (LIKE A$)
3
NAME
STRING LENGTH
STARTING ADDRESS
SINGLE-DIMENSION INTEGER ARRAY (LIKE DA(n))
2
NAME
NUMBER OF BYTES TO NEXT VARIABLE
NUMBER OF DIMENSIONS
DEPTH OF FIRST DIMENSION
ELEMENT 1
ELEMENT 2

Fig. 10-10. The three types of variables handled by the assembled version of Idword. Integer variables and arrays are stored in separate areas of RAM.

is any larger than five letters. PUSH the string starting address onto the stack for safe-keeping. The string length is hidden in register C for later use.

Now the table search begins in earnest. Idword presets the register *B* to the length of A$, and DE is set to the starting address of A$. HL acts as a memory counter, incrementing from the start of the word table to the end. Then the comparison loop begins. Idword checks a character in the word table to see if it is a zero or a comma (ASCII 44). If so, the end of a DATA item is hit, and the next item must be found. If a zero is encountered, the end of the DATA line has been reached and Idword must skip ahead five bytes to reach the start of the next line. If a comma is found, one byte must be skipped (the comma itself) to get to the next item. Then the routine loops back to reset the values of *B* and *DE* for a new comparison. This is because the present comparison is considered to have failed if the end of the DATA item is reached this early.

If neither end-of-item code is found, one more preliminary check is performed. If a period (ASCII 46) is encountered, then the entire table search is considered a failure. Why? Because the last word in the table is a period; if it is found, the table search has reached an end without a successful word match. Idword cleans up the stack a bit, using POP, and returns. N equals zero, indicating a search fail, since *N* was set to zero immediately prior to the USR call.

If none of the above codes are found, Idword performs the actual comparison of words. A letter of A$, as pointed to by DE, is compared to the corresponding letter of the table entry, as pointed to by HL. If the match fails, a quick loop occurs which advances HL past the remainder of the table entry up to the end-of-the-item marker, either a zero or a comma. Then, Idword loops back to reset DE and B for another check.

If the comparison of the two letters is successful, however, B is decremented and the next two letters are compared. This loop continues until B runs out to zero. When this occurs, Idword knows that every letter in A$ is in that table entry. That is not enough; what if A$ is just a small part of that entry? (For instance, the input S passes this test if the table entry is SE, but the two are not the same command.) One more test is done. The next character in the table entry is checked. If it is an end-of-item code, zero, or comma, then the match is identical. If not, Idword jumps to the loop that skips the rest of that entry and loops for another item.

If it is an identical match, the final task is to read the very next item in the word table, which is the ID number, and store that numeric value in variable *N*. Idword advances to the next item according to the kind of end-of-item code it encounters. Then it calls the address 1E5AH, which refers to a routine in the TRS-80 ROM. This routine takes any ASCII-coded number pointed to by HL, converts it to a binary value, and stores it in DE. Once this is done, the stack is adjusted and the value in DE is temporarily stored on the stack.

Finally, Idword must find the variable *N* in memory so that it can change its value to that of the binary-coded ID number. A search is performed, similar to the earlier search for A$. When *N* is found, POP the value off of the stack and store it in the fourth and fifth bytes of that variable. Idword is finished, so it returns, ending the USR call.

There is one inconsistency in the routine that nevertheless does not affect its performance. The routine searches every item in the table for a match—including the ID numbers. (Technically, it should skip these.) If a player types in a number instead of a word, there is a chance that a match might occur. But even if this sort of erroneous match occurs, nothing goes awry. Why not? Idword, upon finding the match, sets N to the numeric value of the next item in the table, thinking it to be an ID number. Since that next item actually is a nonnumeric word, its numeric value (as computed by the ROM routine at 1E5AH) is zero. Thus, a wrong match always results in N equalling zero, telling the calling program that no acceptable match occurred.

Is it worth the trouble to incorporate this routine into the adventure program? Try it and see! Since at least 80 percent of the delay from command to response is due to the word table search, the speed increase of a machine-coded Idword is well worth the small trouble of creating a new line 1 and typing in the simple POKE loop of Fig. 8.

AND FINALLY

For most of you, a hybrid Basements and Beasties is plenty fast for enjoyable adventuring. Some hard-driving programmers will still thirst for precision and efficiency. Such readers eventually consider the ultimate challenge—an adventure program completely in assembly language. For these few hardy souls, some final comments and suggestions are offered.

First of all the challenge of an assembly-code adventure is made tougher by the limitations of one's equipment. Remember that this book assumes that you own a TRS-80 with only 16K bytes of usable memory and tape I/O instead of disk. The problem with this is that editor-assembler programs based on tape I/O are abominable to use when creating machine-code programs of any real size. It is impractical to try to create a program any larger than 1K of object code using a tape-based assembler on a 16K machine.

This limitation can be circumvented by writing the program as a series of assembled modules. The catch is that each module has no way of knowing what addresses to use when addressing another module, unless you explicitly state those addresses in each module. One change in a module can then require changes in many other modules at once. (Disk-based assemblers get around this by providing a program that links the modules, filling unresolved addresses in one module with locations in others.)

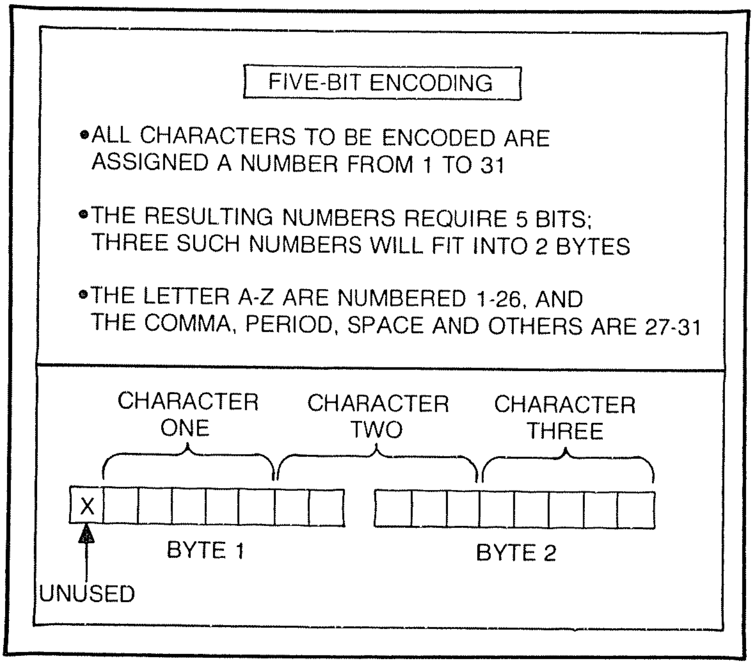
Assuming you are willing to suffer these sorts of discomforts, you must next contend with the problem of housekeeping. BASIC is nice, in that lines of text can be displayed simply by PRINT. What do you do in machine code? When you want to print a line, how do you do it? Or how do you get a command from the keyboard? Fortunately, TRS-80 ROM contains some routines that can be called upon to do some of these menial chores. Check Fig. 10-11 for a few examples.

Another aspect of housekeeping has to do with variables. In BASIC you can say A=1. You don't have to look for a place in memory for A to be stored—BASIC does that. The assembly language adventurer needs to think ahead and set up areas of memory to store all of the numbers that an adventure program needs to maintain. Routines must be written to access these areas.

On the other hand, consider the advantages. Since assembly language is so fast, there are tricks that can save memory that would

ADDRESS	
032AH	The character in A is printed on the screen
28A7H	A message line ending in a zero byte and pointed to by HL is printed on the screen
1BB3H	Up to 255 characters are input from the keyboard and stored in a buffer; HL is set to one less than the buffer beginning
0049H	A single key input is loaded into A; the routine waits until the key is pressed

Fig. 10-11. Some ROM-resident routines that can be called from a machine-language adventure program.


FIVE-BIT ENCODING

- ALL CHARACTERS TO BE ENCODED ARE ASSIGNED A NUMBER FROM 1 TO 31
- THE RESULTING NUMBERS REQUIRE 5 BITS; THREE SUCH NUMBERS WILL FIT INTO 2 BYTES
- THE LETTER A-Z ARE NUMBERED 1-26, AND THE COMMA, PERIOD, SPACE AND OTHERS ARE 27-31

Fig. 10-12. One of many methods for encoding characters for compression. This method, though inefficient, is very easy to implement in assembly language.

be impractical in slow BASIC. The best of these tricks is to use what you might call "compressed descriptions."

What is a compressed description? First, think of all of the memory that room descriptions and object descriptions take up in BASIC adventure programs. If these lines can be stored in compressed form somehow, much space can be saved. A compressed description is one in which text material is stored in encoded form and is decoded only when displayed.

Figure 10-12 shows one method of encoding text for compressing storage; there are many others. In this method the characters in a section of text are limited to only 31 different characters: the 26 letters plus a handful of separators, such as, spaces and periods. To encode a paragraph every three characters is compressed into two bytes of memory, by giving each character a value of from 1 to 31. Since a number in that range requires only five binary bits, three encoded characters require a total of 15 bits, which fits easily into two bytes. Thus, by some voluntary limitation of one's selection of letters, a savings of 33 percent can be achieved.

Chapter 11

Graphic Adventures: The Concepts

After the last pages, you'd think that there is nothing that could be added to the discussion of adventure programming. However, there is a growing circle of microcomputer users that would take issue with that statement. After all, the sort of text-oriented adventure programs that I've examined are merely the forefathers of today's computer-gaming genre. Newer types of adventure programs are swiftly moving into the market, replacing the old. By far, the predominant form of these newer games is that of the *graphic adventure.*

Why is this true? At least one reason is to take advantage of the architecture of the modern-day microcomputer. In the earlier days, computer use was time-shared and terminal-based, and the terminal was not necessarily even a CRT, but possibly a teletypewriter. Adventures were of necessity text-oriented. Today's small computers are for the most part designed for quality graphics and cursor-control. (Most other types of microcomputer games take advantage of the versatile screen; why not adventures as well?

COMPARING ADVENTURES

There are several differences between the structure and operation of the text-based and the graphic adventures. Here are a few of the differences. If you remember the earlier chapters, you may think of others.

First, there is a less stringent memory requirement. What was the real memory-hog in the older-type game? Clearly it was the space required for text: room and object descriptions, special messages, and the word table. In the graphics form of the game, the required text is lessened dramatically. Room descriptions are entirely replaced by graphic representations of the rooms. The object descriptions, likewise, are replaced by representative characters that appear on the screen to indicate each object. Not as many special messages are required. Commands are entered by single key strokes, replacing the need for a word table and its parsing routines.

The result, obviously, is that more free memory means more possible rooms. Graphic adventures usually boast scores more rooms than competing text games. As an example, compare Basements and Beasties with the sample graphics game Mazies and Crazies. The first game has only 20 rooms; the latter, a total of 90! More rooms definitely increase the interest of an adventure game.

Second, graphic adventures are played in real time. In other words, the passage of precious seconds is a real factor. The older games are *command-driven*; things happen in response to each entered command, but all action freezes until the next command. The newer games are clockdriven; action goes on as you watch, and you can choose to act or react at any time.

Again, the interest-level is increased. If a graphic creature is attacking, you cannot simply walk away and take a snack break; you must fight or die! Thus a new element, quick eye/hand response, becomes crucial to success in the game.

Third, there is an emphasis on battle. In the older games, creatures are largely obstructions to travel, except for the occasional tenacious creature that might come along. In graphics games, all creatures are tenacious, hostile, and battle the adventurer to the death. In Basements and Beasties, battle was determined by random numbers; in Mazies and Crazies, opponents possess strength levels that affect the course of the conflict.

As a result, the new adventurer must be a wiser fighter. His victory is no longer dependent on the computerized flip of a coin. He must take into account the types of weapons he has available to him, his own strength-level, how far away food and medicine might be, and the strength of his opponent.

Fourth, the details of the scenario are subject to random initialization. The placement of most objects is determined at run time, not at creation of the program. Thus, while the actual room map and room details remain the same, the location of treasures and creatures is new in every game.

Finally, from a structural standpoint, graphic adventures are easier to construct. Executive does not have to interpret many, many possible variations of commands. Fewer commands mean fewer handlers. Since objects are represented as special characters on the screen, the screen itself can be considered as a form of information storage; thus, fewer data need to be stored in arrays.

One case in which this generalization does *not* hold true is that of the actual screen handling. In Basements and Beasties, all you had to do was PRINT. Now you must be concerned about how to represent objects, creatures, and treasures graphically. You must be careful about how you move them and be aware of what happens when two moving things cross paths. You must write a screen update routine that does not take two minutes to draw a newly entered room. As you had to become a master at text handling (command parsing, word table look-up, text access) in text adventures, now you must become a master at graphics (what to POKE and where) in graphic adventures.

DISPLAYING A ROOM

The heart of graphic adventures is the video display of the room. No longer can the programmer merely print a verbose, image-provoking description; he must really paint the room on the screen. Walls, vertical, horizontal and diagonal, must be plotted, with doors in predetermined locations, and each article, animate and inanimate, must be represented by a defined symbol.

In Mazies and Crazies, as in many other graphic adventures, the screen is divided into two separate areas. The leftmost, larger area is called the *action field*, and the area to the right is the *status field*. It is in the action field that the room is drawn and the adventurer (or mazer) interacts with displayed items. The status field is an all-text area, and displays frequently-updated information pertinent to the playing of the game. Managing to paint your picture and print your status data in separate areas with no overlap is part of your ability as an accomplished programmer!

Let's look at the action field in a bit more detail. Figure 11-1 shows a typical screen display in Mazies and Crazies, with the two fields. The action field takes up 88 percent of the screen. The current room is shown as an open square of graphic blocks, broken by occasional blanks that are doors to adjacent rooms. Within the frame are assorted walls (called features of the room), again painted using graphic blocks. Finally, there are several standard characters scattered about, each designated as a different type of object.

The first question that may be asked is this: why use such coarse resolution for the graphics? The walls are plotted on a grid

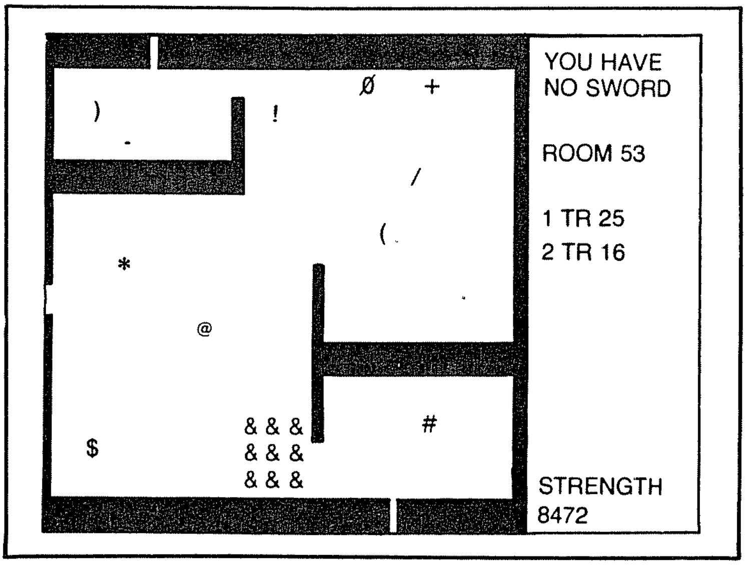


Fig. 11-1. Typical screen display for Mazies and Crazies.

that is 64 by 16, the same layout that is used for the display of alphanumeric characters. Yet the TRS-80 supports a graphics grid with the higher resolution of 128 by 48, using the BASIC statements SET, RESET, and POINT. Wouldn't high resolution improve the appearance of the action field?

The reason for the choice of low-resolution graphics is to allow easy interaction between graphics and alphanumeric characters. Using a single alphanumeric character to represent an object is very easy to handle, especially for moving objects. Characters, however, adhere to the 64 by 16 grid. For instance, if there is a door that is only one unit wide in high-resolution mode, the single-character objects aren't able to fit through it anyway. There is efficiency, also, in treating everything in the display field in the same way. Objects and walls alike are drawn by the POKE command and examined by the PEEK command. Overall, it is much easier (and quicker) to draw the action field in this way than to create high-resolution multipoint graphic symbols for each object and to move them about.

All walls are plotted by *whiting out* successive character-locations on the screen. To do this POKE the graphic code number 191 into each desired screen location in screen memory (graphic character 191 is a white rectangle). With a grid of 64 by 16, there are 1024 possible character locations, 896 of which are used for the action field. Knowing that screen memory begins with address

15360, it is possible to draw any feature desired with the POKE command.

Although the doors that appear in the outer walls look like simple spaces, they are not. Most of the area within the action field is filled with space characters, character number 32; but doors are represented by graphic character 128. This special character looks blank, as if it is a simple space, so it makes a good door. It is distinguishable from a mere space by the program, using a PEEK to view screen memory and see a value of 128, not 32. In this manner, as the player moves up to and contacts the door, the program can invoke a routine to send the mazer into a new room.

For each object, there is an alphanumeric character that visually represents it. The adventurer is seen as the "at" sign (@) and moves about freely. The creature or opponent is shown as an asterisk (*) and attacks at will. A treasure is represented, naturally enough, by a dollar sign ($).

There are various tools that can be present. The torch is an exclamation point (!) and must either be carried or present in a room to allow the player to see. The number or pound sign (#) is a portal, a mystic doorway which, when touched by the player, teleports him into a randomly selected room. The shield is a left parenthesis, which lessens the severity of wounds inflicted by an attacking creature. The right parenthesis is a bow and the dash (—) is an arrow. The potion, a medicine that restores full strength to the player, is seen as a plus sign (+). A period (.) is a kind of fruit that is stored in selected rooms and nourishes the player. The sword is a slash (/) and is the player's primary battle weapon. A field of fire is suggested by the block of ampersands (&) across which the player can go only painfully. Finally, the zero (0) is a bomb, which can be safely carried, but it destroys either the player or creature who blunders into it. (See Fig. 11-2 for the complete object list.)

Note that there are several ASCII characters that can be displayed by the TRS-80 Models I and III that are not defined above. The ones that are defined were chosen because, to degree, their appearance suggests the objects they represent. Other characters can be defined and accessed if they suggest a useful new object. The angle bracket or greater than sign (1), for instance, may represent a sword that has been broken by an ill-fated attempt to hack away at a shell-backed creature. An equals sign (=) may suggest a poison-dart blowgun. Later you'll see how the structure of Mazies and Crazies allows for easy expansion along these lines.

To the right of the action field is the status field, which keeps the player informed of the progress of the game. The action field is filled

OBJ NO.	TYPE	SYMBOL	CODE
1	Torch	!	33
2		"	34
3	Portal	#	35
4	<Treasures>	$	36
5		%	37
6	Fire	&	38
7		'	39
8	Shield	(	40
9	Bow	)	41
10	<Creatures>	*	42
11	Potion	+	43
12		,	44
13	Arrow	-	45
14	Fruit	.	46
15	Sword	/	47
16	Bomb	0	48
17-48	Treasures	$	36
49-96	Creatures	*	42
49-54	Spiders	73-78	Huge Bees
55-60	Snakes	79-84	Amoebae
61-66	Landcrabs	85-90	Trolls
67-72	Scorpions	91-96	Dragons

Fig. 11-2. Object list for Mazies and Crazies.

with text using the PRINT@ function of Microsoft BASIC, which allows precise positioning of the wording without overlaps or carriage returns that would disturb the action field unintentionally.

The status field may itself be subdivided into four *Windows*, each intended to display a different piece of status data. The topmost is the message window. In it, responses to any given input command are printed, as well as warning messages generated by the progress of the game, such as, the proximity of a dangerous creature.

The next subdivision is the *room window*, which always provides the number of the current room that the action field is displaying. The third subdivision is the *inventory window*. In it, the names of any objects carried by the player are listed, arbitrarily numbered from 1 to 8. The player may carry a maximum of eight objects. The inventory window can be entirely blank, of course, if no objects are being carried. Objects with specific names are listed; general objects, such as the 32 treasures, are indicated by treasure number (1 to 32) prefixed by TR.

The final subdivision is the *strength window*. Here the running strength level of the player is continually displayed and updated. Since this level is changing by the second, the strength window is of all four windows the most rapidly updated. The player begins with a strength of 10,000, but this amount wanes with time, with motion, with exertion in battle, and most dramatically with wounds sustained in battle. Only the consumption of fruit or a potion can raise the strength level to a safer amount.

SINGLE-KEY COMMANDS

Gone are the days of one - and two-word command sentences that require interpretation. All commands in a graphic adventure are now entered by single keystrokes. This is consistent with the nature of the newer game as a real-time program. It makes sense to speed up command-entry time in a game so much more oriented towards quick decision-making and response. Then, too, it would never do to have the whole real-time program stop in midstream while the player types in a command word.

Using the INKEY$ function of TRS-80 BASIC, Mazies and Crazies can scan the keyboard as it loops through its realtime Executive. Any key on the keyboard can be used to directly call some handler, based on the character number generated by pressing it.

Figure 11-3 shows the commands available in Mazies and Crazies. Again, as in the case of object assignment, commands and their associated handlers may be added as desired; the structure of the program allows for lateral expansion.

Obviously, the primary function to control by command is motion. The four arrow-keys of the TRS-80 serve well in this capacity. The player cannot move diagonally, except as the result of two consecutive keystrokes. Later you'll see that the creatures are not thus limited and can move diagonally as needed to intercept the fleeing player.

Using the arrow keys, the player can move about, one step at a time. Obviously, he cannot pass through walls (except through doors). As he moves, the program is continually checking the path ahead. Is there an obstacle? In most cases an obstacle in the player's way simply prevents motion. In two cases, the bomb and the portal, motion is radically affected. Contact with the bomb is fatal. One final special case is fire. The player can pass through fire, stomping it cold as he goes, but it weakens his strength level by several points.

If the player attempts to leave by a door, his contact with the graphic-character 128 alerts the program to access a certain table. In this table the program can locate the end destination of the door, and

Fig. 11-3. Single-key command list for Mazies and Crazies.

KEY	HANDLER /COMMAND
Arrows	MOVE the Mazer
T	TAKE an Object
1-8	DROP an Object
F	FIGHT with Sword
S	SHOOT the Arrow
Q	QUIT or Score

the features of the new room so that it can be properly drawn. If the torch is not being carried nor lying in the new room, however, motion in that direction is impeded, with a warning that it is too dark to enter the room.

After mere motion, the mazer wants ways to affect his environment. In order to carry out his primary goal—the hoarding of treasures—the mazer must be able to pick up objects and later drop them in room 1, which is the home base for the maze. This implies the next two commands.

Pressing T allows the player to take an object nearby, assuming he is immediately adjacent to it. The program scans his surroundings (starting at a point above and to the left of the player) and takes the first portable article it finds. (Creatures, fire, walls, and such are not portable.) Accordingly, if there are several portable objects nearby, the player picks up one of them according to the scan sequence. The only real limit is the maximum of eight placed on the player's inventory. As in other adventure games, it reduces the challenge if the player can carry as much as he wants.

Conversely, the mazer may drop any one of his burdens selectively. This is accomplished by pressing one of the numeric keys from 1 to 8. In the inventory window of the status field, each item carried is tagged with a number between 1 and 8, and each item can be dropped by pressing the key matching this identifying number. Dropped items are deposited in a circle around the player. If, for some reason, there is no room around him for all items that he drops (say, if he is in a corner or already closely surrounded by objects), a warning message in the message window alerts him and he is prevented from dumping his load.

The next activity that the player pursues is battle. He has two primary weapons, the sword and the bow. The sword is used by the F key, representing the command FIGHT. Assuming there is a creature close by and that the sword is being carried, a percentage of the creature's strength-level is shorn, with a lesser depreciation of the mazer's own energy. Attempts to swing away at a creature too far away or in a room with no enemy, or attempts to fight with no sword, all generate appropriate responses in the message window.

The S key allows the player to shoot an arrow at a creature. (He must, of course, be carrying both the bow and the arrow.). Alas, the arrow simply bounces off of a creature that is stronger than a given level. Then, too, in many cases the arrow misses the mark and clatters off into a corner of the room, where it must be retrieved to be used again. If it does kill the creature, it does so swiftly and without a drain on the player's own strength. The arrow can be plucked from the body of the creature for reuse.

Finally, there are auxiliary functions that can be helpful in the process of conquering the maze. The Q key invokes the Quit function, which provides an instantaneous evaluation of the present score. The score is based on the number and types of creatures slain and treasures recovered, less penalties for deaths experienced (since all deaths are recoverable by resurrection). The player can choose to terminate the game then and there, or to go on as if the Q key has never been hit. Incidentally, the Quit function makes use of most of the status area. The area is refreshed properly if the player chooses to continue the game.

The programmer may choose to add other single-key functions as far as memory allows. H may invoke a Help function, which lists the objects and their symbols in the status area, or perhaps lists all commands. Perhaps an R key could call the Rest handler, which could stave off death by allowing the player to regain strength in sleep.

DOORWAYS TO WHERE?

In Basements and Beasties, a sizeable portion of code is dedicated to the support of the travel table. A scenario, be it in text or graphic adventures, is merely a set of rooms connected by an orderly list of defined doorways as connecting paths. Without doorways, there is no relationship between the rooms and no real sense of an actual, mapped, travel experience.

How Mazies and Crazies handle doors is at once similar to and yet different from the method used in Basements and Beasties. Let's compare the two approaches.

First of all the number of doors in a room in the old game is limited by the number of directions in which the adventurer can go. He has ten possible directions—the eight compass points plus up and down—and usually uses only a few of these. Remember, too, that there is no such thing as motion within a room; motion always takes the player through a door into a new room.

In the new game doors can be defined at any of the 896 locations that make up the action field. This is made possible by the wide amount of intraroom motion allowed the player. The penalty of this new liberty is that the new form of the travel table needs to specify the exact grid location of each door. That is, each door has an X or horizontal coordinate and a Y or vertical coordinate, the two of which plot the door on the action field.

Then, too, it is not enough simply to specify the room destination at which the player arrives if he uses a given door. When the new room is drawn, where should the player be placed? Surely not randomly in the middle of the floor! No, the new travel table, whatever form it takes, must specify both the final room number and the coordinates (X, Y) at which the player is deposited. These coordinates should place the player just inside a corresponding door within the new room. Door coordinates and destinations must be decided upon systematically and logically.

In place of a travel table, Mazies and Crazies has an explicit string stored in program memory for each room. These room strings contain coded information that specifies the coordinates of doors in the room and their destination coordinates. Room strings also contain data to plot the various features when that room is displayed.

REACHING THE GOAL

Ultimately, the player of Mazies and Crazies has three goals: (1) to stay alive, (2) to collect all available treasures, and (3) to slay all creatures.

Many factors in the game hinder reaching these goals. For instance, even granting the lack of an active opponent, the player can starve to death if he does not locate some food. Treasures are plentiful, but they are scattered throughout the 90-room scenario (and 90 rooms is a bunch!). Creatures are not easily slain, and they "home in" on the player to attack. No, obtaining a high score in this sort of game is no easy trick.

Fortunately, there are some aids that are designed into the fabric of the game. Consider first that the game begins in room 1, the home base, and two helpful tools are placed there for the mazer: the torch and the sword. In addition, no creatures ever stray into rooms 1, 2, or 3; so the mazer always knows that in a pinch he can run for these rooms.

What about hunger and exhaustion? The player's strength level decreases while he rests and drops more quickly as he moves. Two means of sustenance are provided. First, in certain selected rooms there is a piece of magic fruit that regenerates itself after the room is vacated. The strength level of the consumer is raised by a percentage when he eats. The player, as he finds these rooms, should keep track of them, and never stray too far from one of them.

In addition to the fruit, there is a potion, a sort of health medicine that pops up in a room from time to time. When taken, the potion restores the player to his full health of 10,000 strength points.

The potion vanishes, to appear randomly in some other room. The player may find the potion and not use it, but remember the room in which it is located.

Now, lest the player get too lazy, he must remember the following warning. Creatures like food and medicine, too! That's right; if a creature is in the room where your fruit supply is, he can eat it just as easily as you! The same goes for the potion, which then promptly vanishes for some other room. This can be frustrating, but who says adventure gaming is a breeze?

Retrieving treasures is a fairly easy task, remembering the usual limitations. You have an inventory limit of eight, and this really means only about six treasures a trip, since you'll need to carry the torch to see, and you are a fool if you leave your Sword behind. All treasures need to be dropped at home base, room 1, in order for them to count towards your score.

Killing the creature is not so easy. To make things easier, there is never more than one creature in a room at a time. You'll find that one is enough! The beasts range in size from a deadly spider to a huge dragon, and the larger creatures start off with a strength-level greater than your greatest. They attack without provocation and repeatedly. They can move diagonally, while you can only move horizontally or vertically.

You are doomed without weapons. You have your sword, which inflicts a wound on the enemy proportionate to your strength level. The beast must be very close for you to hack him with the sword. You have a bow and arrow, which is more accurate the closer the creature approaches. Even the closest of proximity does not assure that the arrow will hit him. In fact, arrows bounce right off the stronger creatures until their strength is worn down a bit.

There are some defenses as well. First, there is a shield, which limits the wounds which a creature can inflict, somewhat. There is the bomb, which can be dropped in the creature's path. If the enemy steps on it, he is blown to bits—so are you, if you are so clumsy as to step on it. Finally, if the urge to turn yellow reigns supreme, there is the portal, which can quickly be dropped, stepped into, and used to whisk the player off to some random and (hopefully) safer room.

ROOM STRINGS

In Basements and Beasties, each room is associated with two types of information. First, there is a room description block that matches a block of text to each room, which serves to describe that room in detail. (In that same block are the short-form names for the rooms that are used after the first visit.) Second, there is the travel table, which defined the end destination resulting from travel in any given direction within a selected room.

In Mazies and Crazies, these entities are superseded by different sets of information. Travel data is replaced by door information, and the text descriptions are replaced by codes to plot a visible room on the screen. Both of these types of information are stored in what is called a room string.

Figure 11-4 shows the essential structure of a room string. Each room has its own room string, which can be broken into two sets of substrings. The first set is a group of door substrings, each of which consists of numerical codes defining the location and operation of all doors in that room. The second set is a group of feature substrings, each of which contains numerical codes used to create walls as well as other room features, such as, a patch of fire or a piece of magic fruit. Separating the two sets of substrings is a dash. This separator is always present, even in the hypothetical case in which there are no subsequent feature substrings.

Let's consider the door substrings first. Remember you must keep track of three pieces of data for each door in your graphic adventure. These are the screen location of the door in X and Y coordinates, the number of the room to which the door leads, and the screen location where the player is plotted when he arrives at the destination room (in X and Y coordinates).

How much space does this data take up? In the case of the X and Y coordinates, X is an integer from 0 to 55, and Y is an integer from 0 to 15. For Item 1 you need four digits to represent X and Y together; four more digits from Item 3 raise the total to eight digits. The room number in Item 2 is an integer from 1 to 99; these additional two digits result in a total door-substring length of ten digits.

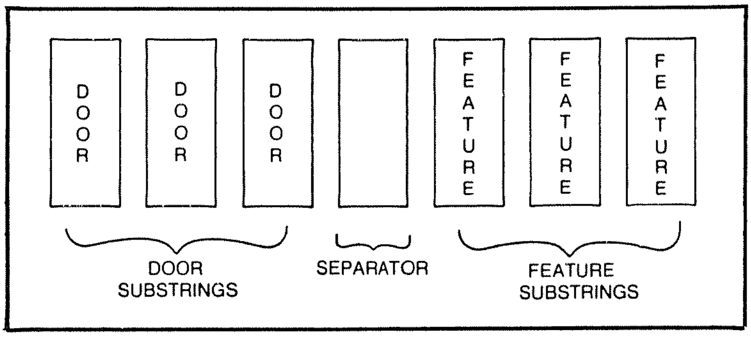


Fig. 11-4. Components of a room string.

CHARACTERS	DATA
1 - 2	Door X-Coordinate
3 - 4	Door Y-Coordinate
5 - 6	Destination Room
7 - 8	New X-Coordinate
9 - 10	New Y-Coordinate

Fig. 11-5. Components of a door substring.

Figure 11-5 demonstrates how the door substring is divided. Since every door substring is ten numeric characters long, there is no need for some sort of character to separate them. Whatever routine accesses these substrings knows to count in multiples of ten characters to move from substring to substring without error. If the routine sees a numeric character, it knows that a new door substring is present; if it sees the dash separator, there are no more doors in that room.

Walls and distinguishing features not only help to identify individual rooms, they increase the challnge by making motion in a room more difficult. Also, certain special features, such as, the magic fruit, require data to specify location.

There are six kinds of features available in Mazies and Crazies to adorn a room. The first four are walls, available as horizontal, vertical, and positive, and negative-slope diagonal lines. The fifth feature is a field of fire, which can be placed to block a door if desired. The sixth feature is the magic fruit.

In plotting these six features, different sorts of data are required. In the case of the first four features, which are straight lines, an interpreting routine must be furnished with a number specifying the type of line, a pair of numbers giving the starting X, Y coordinates, and a number telling the length of the line. In the case of features 5 and 6, no length is required, but the feature type and X,Y coordinates are still necessary. Thus, there are two types of feature substrings—a long one and a short one. The long one requires seven characters, and the short one only five.

Figure 11-6 shows how the feature substring is divided, both in its five and seven-character forms. The routine that does the plotting knows how long the substring is based on the beginning feature type number and can thus find the start of the next feature substring without any dividing marker.

For ease of entry and location, the 90 room strings are stored on separate program lines in a block of memory by themselves. In order to allow easy access to the strings, they are equated to the elements of a 90-deep string array. The use of string array names does not recopy these explicit strings into free memory; rather, it sets up a list of addresses that point to these strings in program space. About 200 bytes of string-array variable-pointer space is required, but the result is worth it—rapid access to the strings for the quickest possible action field refresh.

THE 90-ROOM MAP

Just as a map or drawing of the interrelations between rooms is necessary in Basements and Beasties, so must a map be drawn for the scenario of the graphic adventure; however, it makes no sense to label the interrelated pathways by the motion indicators N, S, E, and so on. Compass points are no longer relevant. Now doors are in specific screen locations in the action field.

Drawing a detailed scenario map in this case becomes impractical. It is still helpful, however, to map out the rooms and their pathways without regard to the specific locations of the doors, simply to show the end room destinations. Later, the exact door coordinates can be chosen as desired to fit the rough layout of the map.

CHARACTERS	DATA
1	Feature Type
2 - 3	Plot X-Coordinate
4 - 5	Plot Y-Coordinate
(6 - 7)	(Plot Length)

TYPE #	FEATURE
1	Horizontal Line (-)
2	Positive Slope (/)
3	Vertical Line (|)
4	Negative Slope (\ )
5	Fire (field of nine &'s)
6	Fruit (single .)

Fig. 11-6. Components of a feature substring.

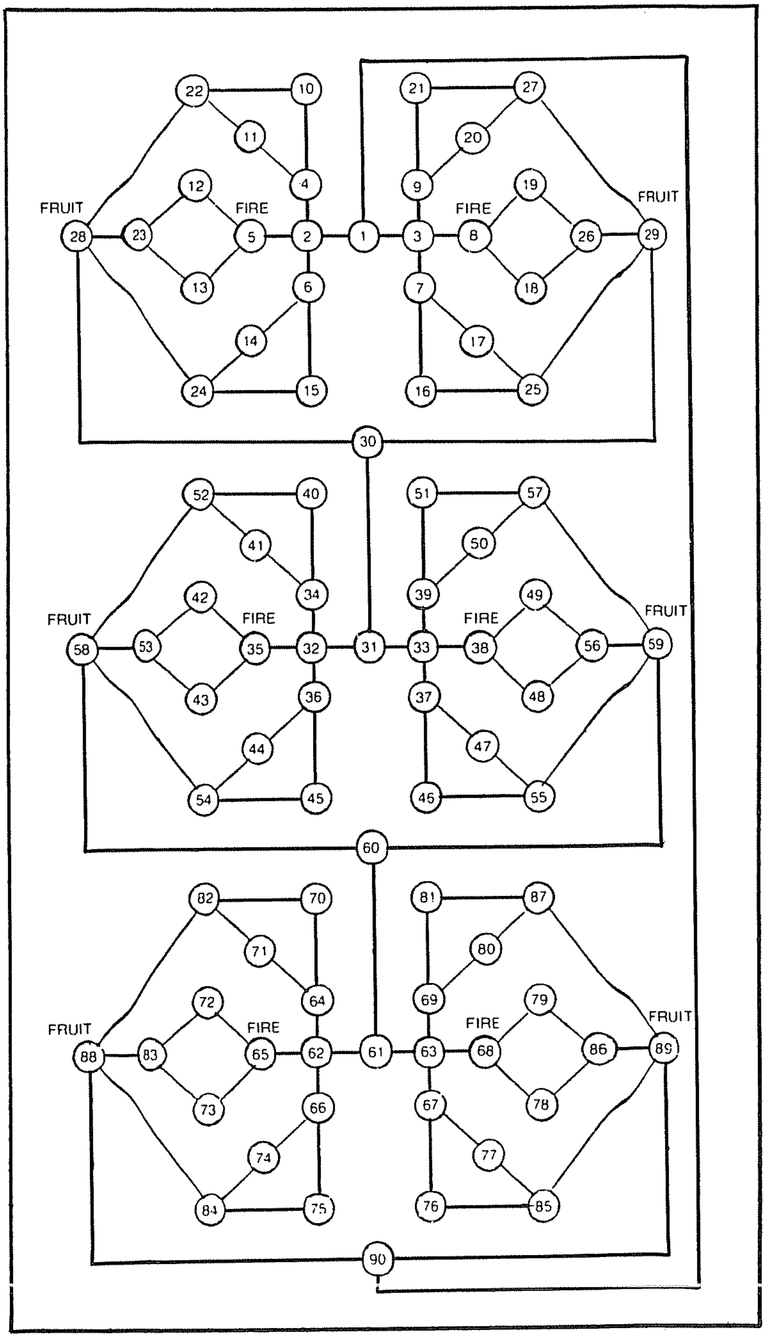


Fig. 11-7. The complete Mazies and Crazies scenario map.

Figure 11-7 provides the map for Mazies and Crazies. Note that the positions of the connecting lines correspond roughly to the final coordinates of the doors. See also that the locations of special features, such as patches of fire and piles of fruit, are marked for inclusion in the room strings.

The symmetrical form that this map takes was chosen for ease of design. You will find that the general appearance of rooms in the action field repeats the symmetry of the map. This is, of course, arbitrary, being based on the room strings. Ideally, each and every room is designed to look distinctly different.

The same rules that go for scenario mapping in text-type adventures also go here. Note the number of careful branches. They are intended to limit options so that each branch chosen is followed. The player finds new layers of rooms each time he leaves one of the root nodes by a different door. This maximizes the suspense of the game.

DOWN TO SPECIFICS

Now you are familiar with the general attributes that make a graphic adventure such as Mazies and Crazies a different game than the text-oriented programs. In several ways it is a much easier program to write. Certainly, the executed code is shorter. For all its simplicity, the graphics make it an inviting game.

Now you are ready to move on to the specific details of Mazies and Crazies from the assignment of variables to the structure and operation of the handlers and subroutines themselves.

Chapter 12

Graphic Adventures: The Segments

Let's start with program structure. As in the text adventures, a high goal is to produce code that is easily followed and easily expanded or modified. BASIC isn't that sort of language by birth; so it is up to the programmer to supply the structure.

Figure 12-1 shows the program's organization. Note the close similarity to the organization used in Basements and Beasties. Mazies and Crazies is notably simpler.

The first program segment is the Initialization section, designated in the area from BASIC lines 0 to 99 (though, of course, it only uses a few of these). In this section all arrays are dimensioned, and variables are preset. Note that the locations of objects, such as, treasures or creatures, are randomly assigned in this section. This, of course, is in distinction to text-type adventures that preassign these locations according to a strict lookup table.

Next comes the Executive loop, from lines 100 to 199. This section is called a loop because the program spends most of its time circulating through this code. Even when the player is not selecting specific commands, the program loops through this segment, updating the player's strength level, moving a creature (if there is one nearby), and otherwise maintaining the status of things. Mazies and Crazies deserves its description as a real-time game due to the continuous operation of the loop.

As in Basements and Beasties, player commands are serviced by the invoking of specific handlers, tailored to bring about whatever

LINES	PROGRAM SEGMENT
0-99	Initialization
100-199	Executive Loop
200-499	Handlers
500-599	Display Section
900-999	Subroutines
1000-1099	Room Strings

Fig. 12-1. Program organization of Mazies and Crazies.

effect is commanded. Handlers reside anywhere from lines 200 to 499 in the program. Generally speaking, there is one handler per command, each of which is called by the pressing of a given key. One special addition required is the display section, which can be found from lines 500 to 599. This section is called whenever the mazer enters a new room. It replots the walls and other features of the room, and indicates the presence of any objects located in that room. This routine makes heavy use of the information contained in the room strings that are stored later.

Next is the subroutine section, from lines 900 to 999. These are subroutines called either by the Executive loop, the display routine, or individual handlers.

The final (and largest) of the program segments is the room string section, delimited by program lines 1000 and larger. The room string for each room resides in a single program line, such that the strings for rooms 1 to 90 are on lines 1000 to 1089. Each line sets a string array element equal to the room string; the room string section is really a large subroutine, called by the initialization section, which simply initializes variable pointers for each access to the room strings. Accordingly, the last line of this section is terminated by a RETURN statement.

ALL OF THE VARIABLES

Mazies and Crazies is a simpler game to maintain than the text adventure games. The list of variables it needs for housekeeping are shorter. Figure 12-2 provides the variable list.

There are 96 objects in the scenario, including 16 tools, 32 treasures, and 48 creatures. For each of these objects, a record is required telling in which room each resides. The room location is stored in the elements of array L(n), wherein n is the object number from 1 to 96. (See the object list in Fig. 11-2). Similarly, the player himself lives in a given room, and this location is stored in L(0).

The contents of the elements L(0) to L(96) can be an integer from 1 to 90, specifying the room location. Also, if a creature has been slain, his L(n) value is set to 0; Room 0 is the grave. Objects that are carried are in the player's carry-sack and are assigned a location value of 91.

Several other values are required for program operation, but it saves some memory to assign these to further elements of the array L(n). Why is this so? The reason is that for each different variable set up by a program, a variable pointer is created. You save a few bytes by making the most out of an already created variable array.

Inanimate objects are assigned positions in the action field largely on a random basis (as you'll see later). This is not true of the player and his opponent creature (if there is one in the room), since both must move above freely in a comprehensible manner. For this reason a record is maintained and updated of their positions in the room. These position records are stored in X,Y form, wherein the X coordinate refers to the horizontal character position from 0 to 55, and the Y coordinate refers to the vertical character position from 0 to 15. L(97) and L(98) save the X,Y position of the mazer, and L(101) and L(102) save the creature's position. Obviously, if there is no creature in the room, the contents of L(101) and L(102) are irrelevant and ignored.

Next, a couple of facts concerning the player are required. The strength-level of the player begins at a value of 10,000, and is continually being affected by battle, exhaustion, or consumption of nutrients. Element L(99) keeps track of the player's running strength. The player may fail in his attempts to slay the foul beasts of

Fig. 12-2. Variables list for Mazies and Crazies.

VARIABLE	APPLICATION
L(0)-L(96)	Object Room Number
L(97)	Mazer X-Coordinate
L(98)	Mazer Y-Coordinate
L(99)	Mazer Strength-Level
L(100)	Mazer Deaths
L(101)	Creature X-Coordinate
L(102)	Creature Y-Coordinate
L(103)	Creature Strength-Level
L(104)	Creature Number
L(105)	Attack/Retreat Flag
L(106)	Retreat Counter
L(107)	Counter
R(1)-R(90)	Room Strings
C(1)-C(8)	Carry-Sack Contents
TA$	Inventory Name String
TB$	Creature Name String
CRS	Current Room String

the maze, dying in some obscure room. Since these deaths count against the player's final score, L(100) keeps a tally of them.

Most of the remainder of the L(n) elements deal with the creature that might be in the room with the player. L(103) reveals the strength level of the creature, which varies as widely as the player's own. The creature number is stored in L(104). If there is no creature in the room at the time, this value is set to 0, telling all routines to ignore the values of other creature-related array elements.

The creature in the room always operates in one of two modes: either attack or retreat. In attack mode it heads directly toward the players; in retreat mode it heads directly away. Element L(105) toggles between 1 and −1, to indicate attack or retreat mode respectively. The creature operates in attack mode until it contacts the mazer or until the mazer fends it off with an arrow or the sword. It then remains in retreat for a random number of steps, as stored and decremented in L(106), the retreat counter. When this counter reaches zero, the creature returns to the offensive.

(Incidentally, you might note that all of these variable values assume the presence of no more than one creature in a room. This is all very proper, since the initialization section is written such that the 48 creatures are separated only one to a given room. The 32 treasures may be divided up between rooms in any old way.)

The final element L(107) is used as a timer. The Executive loop reiterates fairly quickly; to decrement the strength levels both of the creature and the player at this rate is too rapid. Thus, the loop is written to weaken the opponents by one degree every ten loops. L(107) operates as a divide-by-ten counter, being refreshed to a value of 10 each time it counts down to zero.

The next variable array is actually a string array, R(n). The characteristic string-designator, $, is left off, since the statement DEFSTR is used in the initialization section to define R(n) as a string array. This saves one byte per line in the 90-line room string section. In that section each of the elements R(1) to R(90) are set equal to an explicit room string present on the program line. This arrangement does not make use of extra memory space used for strings in high memory. Rather, variable pointers are set up to address the room strings where they reside in the program. In this way, characteristics of a given room can be found simply by reference to the appropriate element R(n), a quick method and an equitable trade-off.

The player can carry up to eight items in his carry-sack. You already know that the carried objects are assigned a location value of

91 in the L(n) array. Handling the carry-sack is vastly simplified if a list of its contents is also kept current. The array C(n) performs this function. In each of the elements C(1) to C(8), the object number of a carried object is recorded. Routines governing the taking and dropping of objects dictate that in the case of fewer than eight carried objects there are no gaps in the list. Thus, objects picked up are recorded in the next available element of C(n), and all unused elements are set to zero. If four objects are carried, for example, their object numbers are stored in C(1) to C(4). The remaining elements, C(5) to C(8), have values of zero. The subroutine that updates the inventory window of the status field makes use of C(n) to list the carried objects.

When the inventory window is updated, the object numbers in C(n) must be translated into object names that are meaningful to the player. So, an inventory name string is set up with a designation of TA$. This string contains 16 six-letter names, one for each of objects numbered 1 to 16, the tools. (Presently undefined objects have a dummy name of "ABCDEF.") The proper routine can extract the correct six-letter name from this 96-character string using some math and string-handling. Treasures, which can also be carried, are not given names and are handled differently in the inventory window.

In similar fashion, different kinds of creatures have identifying names. The 48 creatures are divided into eight kinds, from the weak spider to the terrifying dragon. When one of these beasts is encountered in a room, the routine that handles the message window must have access to a string of text that defines the eight creature names. For this purpose a second text string, the creature name string, is set up with a designation of TB$. This string contains eight eight-letter names for the creatures; names shorter than eight letters use spaces to fill their substring. Again, string-handling and some calculation can extract the correct name.

The final designated variable is the current room string, identified as CR$. The variable CR$ is really more of a convenience. Whenever a new room is entered, CR$ is set equal to the current room string, R(n). Subsequent routines can always refer simply to CR$, without needing to access the specific string array element. Otherwise, every reference to the appropriate room string would require the cumbersome expression R(L(0)).

Having considered the variables used to support the game, let's turn our attention to the program code itself. For the bulk of this description, refer to the complete listing of Mazies and Crazies provided later in Chapter 13.

INITIALIZATION

The initialization section of the program performs the following tasks:

- Displays a welcoming message,
- Establishes the size variables and string space
- Places treasures randomly
- Places creatures randomly
- Defines text strings
- Places tools randomly or specifically
- Presets player values

Initialization begins with BASIC line 2. The game title is printed, preceded by special character 23 to place the screen temporarily in 32-character large-type mode, for appearance's sake. A CLS statement at the end of the initialization section returns the screen to normal mode.

Next, string space and variable sizing is managed. The CLEAR statement is used to reset all variables and to reserve 256 bytes of string space, which is plenty for the nominal demand incurred by occasional string-handling operations and the maintenance of the current room string. (Most strings used in the game are explicitly defined and stored in program space.) The DEFINT statement is used to declare all variables beginning with A through Z to be integers. This saves plenty of memory space and calculation time.

The major numeric arrays, L(n) and C(n), are sized using the DIM statement. Next, DEFSTR declares variables beginning with R to be strings. This is to save space in the case of the 90 explicit references to elements of the string array R(n), which is then sized using DIM. The elements of R(n) are then equated to their corresponding room strings by a subroutine call to the room string section located at lines 1000 and beyond.

Take a look at the room string section starting at line 1000. Notice the structure of each program line, which defines an explicit string. You should be able, with a little patience and some review of the previous appendix, to dissect both the door substrings and the feature substrings, and thereby to get a hint of what any given room should look like. Later, you'll study the code that actually performs this sort of analysis.

Line 4 of initialization scatters objects numbered 17 to 48 (the treasures) to the four winds. The location array element L(n) for each item is set to a random value from 4 to 90. Thus, the treasures may be found in any room other than the home base (room 1) or the two rooms directly adjacent to the home base (rooms 2 and 3). Also, more than one treasure may turn up in any one room.

The same cannot be said for the placement of creatures, which is performed in lines 6 to 8. Because of the way in which creatures are manipulated, it is necessary to limit them one to a room. To do this, a scratchpad string, X$, consisting of one character per room is set up. By accessing the variable pointer for this new string using the VARPTR statement, the memory address of the first character in X$ is stored in variable P. Then, for each of the creatures (objects 49 to 96), a random room number is generated.

The idea is to use X$ to record which of the rooms contain creatures, and to flag those rooms as invalid choices if their numbers randomly come up again during the FOR-NEXT loop. For each randomly selected room number, the corresponding character of X$ is checked to see if it is a space (character 32) and therefore available, or a dash (character 45) and therefore unavailable. If it is a space, the creature is placed in that room, and the space is changed to a dash, to eliminate the room from future selection. If it is not a space, line 8 loops to itself, generating random room numbers until it finds an available room. The location array element L(n) is set to the room number selected. When the room numbers are all selected, the scratchpad string X$ is set to a null length, effectively removing it from active service.

Notice again the use of the expression RND(87)+3, which selects a room number from 4 to 90. Creatures, as well as treasures, are not found in those first three rooms of the scenario.

In line 10 the two text strings are explicitly defined. TA$, the inventory-name string, and TB$, the creature-name string, are created. Note that the spaces in each string are critical, since the proper extraction of a name depends on counting through the string character by character.

Line 14 places certain tools in random rooms, again from 4 to 90. Notice that a loop is not used (as in line 4), because not all tools are to be placed randomly, nor are all objects from number 1 to 16 defined. (If an undefined tool is placed in a room, the action field shows a symbol, such as a comma for object 12, which can be picked up and carried, but serves no purpose. The inventory name is "ABCDEF.")

Next, line 16 selectively places certain tools in room 1. Objects 1 and 15, the torch and the sword, are really necessary for any progress into the maze, and so these are helpfully placed at the home base. Later you'll see that these are dropped back at home base if the player is killed and resurrected.

Finally, the player's location and strength are set. L(0) is set to 1, placing the player at home base. His X,Y coordinates are set to place him roughly at the center of the action field. His strength-level is preset to its starting value of 10,000.

A CLS statement concludes the initialization operation by clearing the screen (returning it to 64-character mode as well), preparing for display of the first room. On the whole, the pregame delay only amounts to four or five seconds.

DISPLAY

Although the next block of BASIC text is properly the executive loop, one of the first things the loop does is plot the current room in the action field. The task of reading the room strings and painting the picture belongs to the display section. This section begins at line 500. It contains the subroutine which we'll refer to as Dsplay.

The steps taken by Dsplay to fill the action field and prepare the player for entrance into the new room are:

1. Reset pertinent variables.
2. Paint an empty frame over the action field.
3. Plot the doors.
4. Plot the features.
5. Plot the player.
6. Plot all objects.
7. Set variables related to the creature, if it's present.

Dsplay starts by clearing L(104), the variable that specifies which creature is present. Unless otherwise updated by step 7 of this routine, the zero value indicates that no creature is in the room. Then, the current room string (CR$) is created by accessing the proper array element of R(n), using L(0) which contains the present room number.

The next step is to paint the basic square frame that forms the basis of the action field. This is done quickly by creating strings and printing them at the proper screen locations. (This is far quicker than looping through each screen location and to POKE individual characters.) This has the additional advantage of erasing anything that was previously within the boundaries of the action field, without having to blank out the entire screen with a CLS statement. Thus, as the player moves from room to room, the status field of the screen remains undisturbed.

First, the top and bottom parts of the frame are drawn, by printing two 56-character strings made up of white blocks (graphics character 191). Then the internal portion is blanked out and the sides of the frame drawn by printing a series of strings consisting of spaces and leading and trailing white blocks.

Now the routine is ready to loop through the room string CR$ to place doors on the screen. The string is examined using the MID$ statement with the variable N pointing to each successive character. N is set to 1 at the end of line 500 in preparation for the loop.

Line 502 begins with a test of the character in CR$ being pointed to by variable N. If that character is a dash, the routine knows that it has come to the end of the door substrings and it can proceed to the next step of the task. Otherwise, it begins to analyze the next few characters according to the guidelines set up in Chapter 11.

Remember that each door substring consists of ten characters, the first two being the two-digit X-coordinate of the door, and the next two being the two-digit Y-coordinate. Using MID$ these pairs of characters are extracted, and they are converted to the original numeric values using the statement VAL. With the variables X and Y set to these values, POKE the door onto the screen using the expression X+64*Y+15360, which converts X,Y notation into a specific screen memory address. Note that the door is represented by the character numbered 128 which, though it looks like a space, is really a blank graphics character distinguishable by the program.

After a given door substring is accessed, it is bypassed by adding 10 to the variable N. Then line 502 is executed again. Line 502 loops to itself in this manner until the dash is encountered. This is the reason that all room strings require a dash, even if there are no subsequent feature substrings. If there were no dash, line 502 would try to read beyond the length of CR$, causing an error.

Once all doors are plotted, line 504 accesses the feature substrings. Variable N is already set to the character position just one past the dash, in readiness for this next scan. Line 504 begins with a test to see if the end of CR$ has yet been reached. If not, the subsequent characters are examined according to the format for feature substrings presented earlier.

For all types of feature substrings, the second through the fourth characters store the X,Y coordinates for the beginning of the feature plot. These coordinates are extracted using MID$ and VAL, similarly to line 502. Then, the specific feature type must be selected. The first character of the feature substring is a digit from 1 to 6. This digit is extracted and used to select the appropriate program line for that feature, using the calculated jump statement ON GOTO. The program lines that handle the plotting of features are located in the block of lines from 506 to 512.

Feature type 1 is a horizontal line. The code on line 506 allows very rapid display of such lines, because it uses the BASIC PRINT@ function rather than plotting each point in the feature with POKE commands. The expression X+64*Y provides the screen location for the PRINT@statement. A string is built of graphic characters 191. The length is determined by examining the sixth and seventh characters of the feature substring; these are restored to true numeric form and used to complete the STRING$ function. Once this string is printed, the pointer variable N that keeps track of the location in the room string is bumped by seven characters, thus skipping on to the next feature substring. Looping back to line 504 checks to see if more features are encoded.

Line 508 handles the other three types of line features. The variable S is assigned as a pointer to track the plotting of each point in the line that reaches screen memory by a POKE. Using the already extracted X and Y coordinates, S is set to the specific screen memory location at which the line starts.

Now, if you think about it, there is an easy way to figure out which memory location is next in the line plot. Consider first the vertical line. Each point in the line is exactly 64 memory locations further along than the previous point. That's because the TRS-80 screen is 64 characters wide. Thus, to plot a vertical line, the pointer S is increased by 64 each time.

What if S is increased by only 63? The next point is below and to the left of the first. If this is repeated, a diagonal line of positive slope is effectively plotted. Similarly, adding 65 to S creates a negative sloping diagonal line.

The feature number which is the first character of the feature substring was chosen in these three cases so that adding 61 to the feature number produces the proper number to add to S. Line 508 extracts the feature number, converts it and adds it to 61, storing the result in variable D. Then a loop is set up, from one to the line length, which is extracted as previously from characters 6 and 7 of the substring. Using the contents of D to produce the proper line angle, POKE the graphic character 191 successively into screen memory locations. When the process is complete, the next substring is sought.

Feature number 5 is a field of fire, as represented by nine ampersands (&). This field is plotted around the point specified by the X and Y coordinates already extracted. Again, to speed things up, the PRINT@statement is used to print strings, rather than using POKE. The field is created by printing three strings of three ampersands each, starting at the point to the left and above the location specified by the X,Y coordinates. This starting location, for the sake of the PRINT@statement, is calculated by subtracting 65 from the center point address.

Line 510 creates the fire field. Three times in a row, a string of three ampersands is printed, increasing the starting location by 64 each time to assure that a perfect three-by-three box is generated. Then the substring pointer, variable N, is bumped by five to access the next substring.

The final feature is the magic fruit, which is simply a period (.) placed at the point specified by X and Y. Line 512 calculates this memory address and use a POKE to place the number 46, the ASCII code for a period, into that location. The substring is skipped and line 504 is again executed.

When all feature substrings are serviced, the player must be plotted onto the action field. Array elements L(97) and L(98) give the player's X,Y coordinates in the room; these are reset by the routine that moves the player. If the entrance is through a standard door, this plotting location is right next to a door in the display. Using POKE character 64 is placed in the proper location, displaying an at sign (@), which is the symbol for the player.

Next, all objects present in that room must be plotted. This occurs in three stages, since there are three distinct categories of objects.

The first objects to plot are the tools, with object numbers 1 to 16. A simple loop checks the location array elements for each tool. Each object that has a location number corresponding to the present room (stored in variable L(0)) is plotted. The character used is the object number plus 32, which produces the symbols described earlier for objects.

Where do you plot these tools? There are no variables that store the X,Y coordinates of these objects. Therefore, they are placed on the screen randomly. A subroutine is called to locate a viable X,Y location for an object. This subroutine is called Randxy, and it is found on line 940. In it random values for X,Y are generated; X equals 1 to 54 and Y equals one to 14, which selects a random spot within the outer frame of the action field. You do not want to plot that object right on top of a feature just plotted! Randxy checks the proposed location using PEEK. If there is a blank space there, it returns; otherwise, it loops, proposing new random locations until it finds a suitable spot.

The same procedure occurs for the treasures, objects 17 to 48. Randxy is used to find locations on the screen for treasures that are located in the current room. This time, however, the number placed into screen memory (using a POKE) is character 36, the dollar sign ($).

Finally, the creatures, objects 49 to 96, are checked to see if any are in the room. The FOR-NEXT loop in this case is left open if a match is found, because there is never more than one creature in a room. A random location is found using Randxy, character 42, the asterisk (*), is put in the address using a POKE.

Assuming a creature is not found, Dsplay is done, and it returns to the calling routine. If a creature is in the room, however, miscellaneous variables are set. L(101) and L(102), as the creature's X and Y coordinates, are set to the random values decided by Randxy. Then L(104) (which is zero if no creature is present) is set to the creature's object number. Dsplay then returns.

THE EXECUTIVE LOOP

The Executive loop is the block of code stretching from line 100 to 130. Depending on the present environment, the Executive loop performs the following functions:

- Refreshes the message window and action field
- Initializes the creature (if there is any)
- Refreshes the status field
- Updates the strength window
- Handles an input command
- Directs the creature's motion or attack
- Handles the creature's death
- Handles the player's death and resurrection

The first task is easy. A subroutine called Clrmes, located on line 920, is called to clear the message window. (Clrmes simply prints strings of blanks at the two message window lines, locations 56 and 120). Then a subroutine call to Dsplay refreshes the action field. For the next step, the variable L(104) is checked to see if a creature is present. If it is a zero, there is no creature and the third step is performed. Otherwise, the strength level of the creature is calculated and stored in L(103). This strength level depends on which of the eight types of creature this one is. Using the creature's object number stored in L(104), the strength level is generated, being from 5000 for the lowly spider to a full 12050 for the fearsome dragon.

The message "BEWARE!" is printed in the message window, along with the name of the creature. The creature name string, TB$, is accessed and the proper eight-character name is extracted using MID$ and some calculations on the object number. The resulting

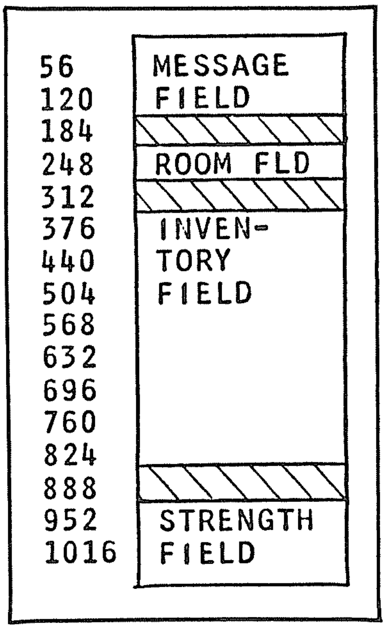


Fig. 12-3. Screen addresses for the status field.

name is printed. Finally, L(105) is set to 1. This variable represents whether the creature is attacking or retreating; a value of 1 indicates an attack, while −1 indicates a retreat.

Creature or not, the next step is to refresh the status field. This is done by calling a subroutine named Status, which is located on lines 900-902. Figure 12-3 shows the layout of the status field with the screen addresses used by the PRINT@statement. Status first prints the headings and appropriate values for the room window and strength window, using PRINT@. Next, a loop scans the inventory array C(n). For each element of C(n) that yields an object number (and thus an object carried), the element number from 1 to 8 is printed. If the object is a treasure, the prefix TR is printed, along with the treasure number from 1 to 32 (which is simply the object number minus 16). If the object is a tool, the inventory name string, TA$, is accessed to extract the proper six-character object name to be printed. As soon as a zero value is found in an element of C(n), the loop is completed. To clean up remnants of a previous inventory listing, a blank line is printed right after the last displayed carried item. Next, the strength level of theplayer, stored in L(99), is printed in the strength window.

The keyboard is scanned for any command input, using the INKEY$ function. Commands fall into three categories: (1) the arrow-key commands that access the Move handler, (2) the numeric-key commands which access the Drop handler, (3) the alphabetic-key commands.

Pressing the arrow keys produce ASCII characters 8, 9, 10 and 91 through INKEY$. The Executive loop checks for this and jumps to the handler Move, at line 260, if so. Otherwise, a call to Clrmes prepares the message window for responses to the next commands. (Clearing the message window between commands makes it easier to tell when new messages are printed.)

The Executive loop jumps to Drop, at line 230, if a numeric key is pressed. If a character comes in that is outside of the alphabetic range, the program skips the command handling altogether.

Line 106 contains the vector list that provides the BASIC addresses for each single-key command handler. There are 26 vector addresses, one for each letter of the alphabet. Letter inputs that are not implemented are simply vectored past the command handling and ignored. The handlers themselves, when they complete their tasks, vector back to one of a few entry-points in the Executive loop.

MOVING THE CREATURE

Assuming no command was input, the creature must now be moved. Line 110 checks L(104) to see if there is a creature in the room. If not, it skips on to the next step. Otherwise, a proposed next position for the creature is generated. This is done by comparing the present X,Y coordinates of both the player and the creature. The new location of the creature is one degree closer to or farther from the player. The attack or retreat variable L(105), which is either a 1 or a −1, is used as a multiplier to determine which direction to move.

The results of a move in the proposed direction are considered before the move is performed. This check is done by examining the contents of the new location on the screen, which is in ASCII-coded form in variable D. There are six cases to deal with:

- Contact with the player
- Contact with the bomb
- Contact with the fruit
- Contact with the potion
- Contact with an empty space
- Contact with an obstacle

In the case of contact with the player, line 112 prints a warning message stating the name of the creature followed by "ATTACKS!"

(The creature name string is accessed as before for the proper name.) Next, to indicate that the player has just been bitten, the location is blinked. This is done by quickly placing the all-white graphic character 191 into the location (with a POKE), then restoring it to character 64, the player symbol. The creature is set to retreat mode by storing a −1 in L(105). Subsequent moves are in the opposite direction from the player. The retreat counter L(106) is set to some random value from one to 20; a creature may flee far, or he may attack again almost immediately.

The player's damage due to attack is calculated. The player's strength level is reduced by one-eighth of the creatures' strength level. In that way the stronger the creature, the more the damage. If the player carries the shield, he is somewhat protected. The location of object 8 (the shield) is checked to see if it is in the carry-sack (location 91). If so, the player's strength level is boosted by 500 points after an attack. On line 114 the effects of the damage are evaluated: did the player die? If not, the next step of the Executive loop is performed. If so, line 128 is executed, handling the player's death and resurrection.

When the player dies a loop is set up, causing the inventory array C(n) to clear all contents, and every object carried is moved to the room where he died. Then L(100), which keeps track of the number of player deaths for scoring purposes, is increased by one. The screen is cleared, and a message is displayed. On line 128 an INKEY$ loop scans the keyboard, waiting for ENTER to be pressed, indicating that the player is ready to continue. To continue the program reenters at line 16, which dumps the player back into home base (room 1) along with the torch and the sword (just to be fair, since he couldn't get far without them).

All of the above occurs if the creature contacts the player. What if it contacts the bomb? On line 116 this case is checked. If so, a loop is set up that effectively causes the bomb to blink in a random pattern of graphic characters; that is, it explodes. This spot is blanked out, and the bomb (object 16) is randomly sent to a room from 3 to 90.

Line 124 handles the creature's death. The creature's symbol on the action field is blanked out by inserting a space with a POKE. A message in the message window proclaims, "AT LAST! IT'S DEAD." The creature's location is set to zero (where only the scoring routine can find it), and then L(104) is set to zero to indicate to the Executive loop that the current room no longer contains a creature. The program slips into line 126, which updates the player's strength level (as you'll see shortly).

What if the creature hits the fruit or the potion line? Line 117 handles these cases. In the case of the fruit, the symbol for the fruit is replaced with a space, again using POKE, and the creature's strength level is boosted by a certain random value. If it is the potion, the symbol for the position is blanked. The creature's strength level is set to 10,000, and the potion (object 11) is sent to some random room from 3 to 90.

What if the creature contacts empty space? If so, it is free to move to that space. Line 118 blanks out the creature's present location, updates the creature's X,Y coordinates in L(101) and L(102), and puts its symbol into the new location—using a POKE. In line 120, the retreat counter is updated if L(105) is a −1; if the retreat counter runs down to zero, the attack or retreat flag L(105) is toggled to a 1, and the creature switches to attack mode. Then in line 122, the creature's strength is updated if the counter L(107) is ready to overflow. (This counter divides the Executive loop iterations by ten.) If the creature's strength runs out, the routine handling its death is executed on line 124.

Finally, what if the creature encounters some sort of obstacle, such as, a wall or an object, other than those already handled? Line 119 sets the creature to attack mode if it is retreating when it hits the obstacle. Creatures rebound from obstacles, as if backed to the wall and desperate. If the creature is already attacking but simply can't see its way clear to get at the player, it must choose an alternate path. It does this by generating up to three possible new X,Y coordinate pairs as random choices. If one of these random locations contains only a space, the program jumps to line 118 and handles it normally. Otherwise, the creature is blocked and must wait for the next iteration of the execution loop to be freed.

REMAINDER OF THE LOOP

Line 126 completes the Executive loop by updating the strength-level of the mazer himself. This update, as is true for the creature, occurs only once every ten iterations of the loop. The counter L(107) is decremented; if it has still not overflowed, the loop begins its next iteration starting with line 104, the command input analyzer. If it does overflow, then it is reloaded with a value of ten. Then the mazer's strength is degraded by a point. If this degradation results in a strength-level of zero, the mazer dies, and the program continues on into line 128, which handles mazer death.

THE MOVE HANDLER

The four arrow-keys are used to invoke the handler called

Move, located on lines 260 through 288. Move must determine the proposed new location and handle contact with various items.

Move begins by deriving the net direction desired based on the key pressed. Of the four arrow-keys, the up-arrow key is the "oddball," generating a character value of 91. The value of the key pressed is still resident in variable D, in which it was placed by the Executive loop. Line 260 changes the value of D to be more consistent, such that the arrows yield values of 8, 9, 10, and 11. Using these numbers, line 262 vectors to other lines that translate the arrow direction into X and Y values: 1 if a positive motion, −1 if negative, 0 if no motion. These numbers are then added to the present X,Y coordinates of the player to generate a proposed new position. The variable Q stores the screen contents of the new position, examined by POKE.

If the motion causes contact with a space, line 274 blanks out the mazer's previous position, updates his X,Y coordinates in L(97) and L(98), POKEs the player symbol onto the Action Field at the new location, and depletes his strength-level by five points to indicate gradual tiring in travel. If this tiring results in a strength-level of zero or less, the player-death routine of line 128 is executed; otherwise the Executive loop is reentered.

If the Bomb is touched, the message "BOOMM!! YOU FOOL" is placed in the message window by line 276. A loop causes the bomb on the screen to flicker with random graphic patterns. Then the bomb is randomly moved to another room, and the player-death routine is invoked.

If the portal is contacted, line 278 generates a random room number from 3 to 90. If the torch is neither in that new room nor in the player's possession (as determined by checking L(1), the torch location variable), the message "NOTHING HAPPENS!" is displayed and the Executive loop is re-entered. Otherwise, the word "POOF!" is printed right near the player in the Action Field. Then after a delay, the player's location variable L(0) is set equal to this random new room, and the Executive loop is re-entered early, such that the action field is refreshed.

If fire is touched, then line 280 blanks the old location, and the player is moved to the new location much like a regular move to a space. However, in the process, his strength-level is docked by a random amount, and the message "FIRE!! YEOWW!!" is shown. As usual, if the player runs totally out of strength, the death-routine is executed.

If a door is contacted (character 128), line 282 checks to see if the torch is being carried. If not, line 288 prohibits motion with the warning, "TOO DARK IN THERE". Otherwise, line 282 sets up a loop to compare the door coordinates with those in the Current Room String Door Substrings. Once a match is found, L(0), the player location, is set to the room number extracted from the door substring. If this new room number is 92, then it is not a destination at all, but indication of death by falling into a pit; line 286 handles this case with a message and a one-way trip to the player-death routine. Otherwise, the X, Y coordinates of the player are set to the destination values extracted from the Door Substring, and the Executive loop is reentered early, when the Action Field is refreshed.

THE HANDLER TAKE

Pressing the "T" key invokes the handler Take, located on lines 210 to 220. The following cases are considered by Take:

1. Is the mazer carrying too much?
2. Is there anything nearby to take?
3. Is the item a Fruit?
4. Is the item a Potion?
5. Is the item a Treasure?

Line 212 begins by counting through the inventory array with variable K, seeking an empty or zero slot to place the new item. If it finds none, the "TOO MUCH TO CARRY" message is displayed and TAKE is done.

Line 121 then scans the immediate surroundings for portable objects. The starting point for the scan is the screen location above and to the left of the player. A nested loop using I and J checks nine locations forming a three-by-three block around the player. If the scan finds a symbol that is not a creature, a wall or fire, or the player himself, the player can pick it up. If the scan completes without finding such an item, a "NOTHING TO TAKE!" message is displayed and the Executive loop takes over.

Line 214 blanks out the object that is being taken, then checks for special cases. If the object is a potion, the mazer's strength-level is restored to 10,000, an appropriate message is displayed, and the potion is randomly sent to some other room. If the object is a fruit, a random increase in strength-level occurs, a "THAT WAS TASTY!" message is shown, and Take is finished.

Line 216 continues special checking. If the object is identified as a treasure, it must be determined which of the 32 treasures it is. A loop looks for which treasures are in the current room; the first one encountered in the loop is assumed to be the one presently being taken.

Treasure or not, the object is added to the player's inventory by setting its location number to 91 and plugging its object number into C(K), where K is the next available inventory slot. A final message acknowledges the transaction and returns the program to the Executive loop.

THE HANDLER DROP

Pressing any numeric character from 1 to 8 invokes the handler Drop, located on lines 230 to 236.

Line 230 sets up a scan area of three-by-three screen locations, just as Take did. This time, however, Drop is simply looking for a blank space at which to drop a given item. If it cannot find a space (character 32), it assumes the player is blocked in close, and refuses to drop any item, warning, "NOWHERE TO DROP!".

Otherwise, line 232 takes the ASCII value of the numeric key pressed (still stored in variable D) and converts it to the value from 1 to 8 by subtracting 48. The Inventory slot referenced by this number gives up its carried object; the location variable for the object is set to the current room number. The last part of line 232 is added so that treasures (i.e., portable objects numbered above 16) will be displayed using the proper symbol.

Line 234 POKEs the dropped object symbol into the located nearby space. Then line 236 essentially compresses the remaining entries in the Inventory array C(n), such that there are no spaces. Drop is done, and the Executive loop is re-entered.

THE HANDLER QUIT

The key "Q" invokes the Handler Quit, which can be found on lines 240 to 246. It performs the following functions:

1. Evaluates treasures retrieved
2. Evaluates creatures slain
3. Evaluates mazer deaths
4. Displays the current score
5. Gives player a chance to quit.

Line 240 sets variable J to zero; J will be used to tally the score. To evaluate the treasures, a loop counts through the location array, finding treasures that reside at home base (room 1). For each safe treasure, points are added to score J based on the treasure number. Treasures are worth anywhere from 17 to 48 points each.

Next, line 242 evaluates the slain creatures. A loop awards points for every creature found residing in "room zero" (having an

L(n) value of zero), indicating that it has been killed. Based on the size of the creature, points awarded range from 9 to 16 points each.

Finally, player deaths are subtracted from the total. Checking L(100), which is used to keep track of these deaths, the player is docked 30 points per death.

Using PRINT@statements, line 244 displays an explanatory message which includes the total score at present. Quit uses most of the status field to print this text. Then line 246 accepts input from the player. A "N" answer to the question of "GIVE UP?" runs a quick loop to blank out the entire status field, and the Executive loop is re-entered at the point where the subroutine Status is called to refresh that Field. An answer of "Y" places the cursor in the upper left corner of the screen and terminates the BASIC program with an End statement. Any other input simply loops line 246; this avoids the frustration of ending a game by error.

THE HANDLER SHOOT

If the "S" key is pressed, the Handler Shoot is executed; it resides in the block of lines beginning at 250. The following contingencies must be allowed for by Shoot:

1. Does the mazer have a Bow?
2. Does the mazer have an Arrow?
3. Is there a creature in the Room?
4. Is the creature too tough for the Arrow?
5. Does the shot miss?

Line 250 checks the first three cases. By referring to the Location array L(n), SHOOT can tell if the Bow is being carried; it will have a Location value of 91. If not, a message proclaims, "YOU HAVE NO BOW!" and the handler exits. Similarly, an absence of the Arrow gives the warning, "YOU HAVE NO ARROW". By checking L(104), Shoot can determine if a creature is in the room. If not, the message is "ZZINGG!!" and the arrow is randomly shot out into the room.

Line 256 places the spent arrow in the Action Field. A call to Randxy get a likely location. The corresponding L(n) value is set to the room number, and the arrow symbol is POKEd into the random location. Then a call is made to a subroutine termed SUBINV, whose task is to remove a given object from the Inventory array C(n); this subroutine is found on line 910.

SUBINV expects the object number of the object to be dropped to be stored in variable A. It loops through the eight elements of Inventory array C(n) until it finds the item. It then deletes the item by copying all subsequent elements of the array backwards by one. C(8) is always set to zero, since it is set to the contents of the always-unused C(9).

After calling SUBINV to remove the arrow from the inventory, the variables that control the retreat mode of the creature are set such that the creature is in retreat. Now, in this present case, there is no creature, but changing these variables does not harm in such a case. Soon, though, this same line 256 can be used to handle the case in which the shot drives the beast away.

Assuming there is a creature in the room, line 252 checks to see how powerful it is. The arrow will bounce off of the hide of the creature if its Strength-level exceeds 5000. If so, the message "BOUNCES OFF HIM!" is printed and line 256 randomly drops the useless arrow elsewhere in the room: otherwise, the arrow may hit the creature.

To determine whether or not the arrow hits its mark, the actual distance from the mazer to the creature is calculated, using the square root of the sum of the squares of the X,Y coordinates. This value may range anywhere from 1 to 55. The distance is subtracted from 81 and used as the basis of a percentage test. A random percentage is compared to the value, such that the closer the target, the greater the chance of a hit. The maximum chance of hitting the creature at any one shot is no better than 80 percent.

If the shot fails, the message "RATS!! MISSED!" is displayed and line 256 again drops the used arrow somewhere in the room. Otherwise, line 254 sets the arrow's location to the room number, and the arrow symbol is POKEd so as to replace the symbol of the vanquished creature. SUBINV is called to remove the arrow from the inventory array C(n). The Creature's Location variable assumes a zero value, and the variable L(104) is zeroed to indicate the room is no longer occupied by a creature. A message proclaims, "GOT HIM! VICTORY!", and the handler exits.

THE HANDLER FIGHT

Pressing the key "F" causes the execution of the handler Fight, which is located from line 290 to 298. The following cases must be handled by Fight:

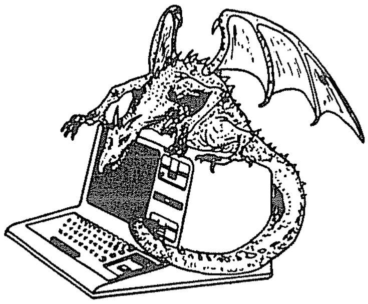
1. Does the mazer have no sword?
2. Is there no creature to fight?
3. Is the creature too far away to hit?

Line 290 checks the location array to see if Object 15, the sword, is present in the player's carry-sack. If not, the message window states, "YOU HAVE NO SWORD." If L(104) betrays that there is no creature in the room, the snide remark "FIGHTING SHADOWS?" is shown.

By subtracting the X, Y coordinates of the two opponents and taking the absolute values, a limit can be set on how close the creature need to be hit with the sword. Line 292 determines that if the creature is not directly adjacent to the player, the message, "MISSED IT! FIE!" is displayed. Then line 298 decreases the player's strength level by 5 percent. If the player's strength runs out entirely, he dies, and the death routine at line 128 is executed.

If the sword hits the mark, line 294 subtracts from the creature's strength-level an amount equal to 20 percent of the player's strength. If this does not totally drain the creature, the message "A GOOD SLASH!" is shown in the message window. Also, a 50-50 chance is generated that the creature is put to flight (switched into retreat mode) by the onslaught. If the swing does empty the creature's strength level, line 296 prints the message "FINISHED HIM OFF!" The creature is dispatched to room 0, the room is flagged as having no creature, and the creature's symbol is blanked out. Whether or not the creature is killed by the blow, line 298 comes into play, draining the player's strength a bit.

Chapter 13



Mazies and Crazies: The Listing

That's all there is to it! All that remains to put Mazies and Crazies in operation is to type it into your TRS-80. The full listing follows for that purpose.

After running "Mazies," you'll find that there are some things you might wish to change. Perhaps the scoring values don't suit you. Maybe you wish there were a few more commands, or some different kinds of creatures. Almost certainly you will want to experiment with the appearance of the individual rooms. As in the text-type adventures, the sky (or at least the memory size) is the limit.

```
          INITIALIZATION SECTION
2 CLS:PRINTCHR$(23):PRINT@532,"WELCO
ME TO":PRINT@590,"MAZIES & CRAZIES!"
:CLEAR256:DEFINTA-Z:DIML(107),C(9):D
EFSTRR:DIMR(90):GOSUB1000
4 FORN=17TO48:L(N)=RND(87)+3:NEXTN
6 X$=STRING$(90," "):P=PEEK(VARPTR(X
$)+1)+PEEK(VARPTR(X$)+2)*256-1:FORN=
49TO96
8 M=RND(87)+3:IFPEEK(P+M)<>32THEN8EL
SEPOKEP+M,45:L(N)=M:NEXTN:X$=""
10 TA$="TORCH ABCDEFPORTALABCDEFABCD
EFABCDEFABCDEFSHIELDBOW   ABCDEFPOTI
ONABCDEFARROW FRUIT SWORD BOMB  ":TB
$="SPIDER  SNAKE   LANDCRABSCORPIONH
UGE BEEAMOEBA  TROLL   DRAGON  "
14 L(3)=RND(87)+3:L(8)=RND(87)+3:L(9
)=RND(87)+3:L(11)=RND(87)+3:L(13)=RN
D(87)+3:L(16)=RND(87)+3
16 L(1)=1:L(15)=1:L(0)=1:L(97)=28:L(
98)=8:L(99)=10000:CLS
```

Fig. 13-1. Initialization section for Mazies and Crazies.

```
              EXECUTIVE LOOP
100 GOSUB920:GOSUB500:IFL(104)<>0THE
NL(103)=(L(104)-49)*150+5000:PRINT@5
6,"BEWARE!";:PRINT@120,MID$(TB$,FIX(
(L(104)-49)/6)*8+1,8);:L(105)=1
102 GOSUB900
104 PRINT@1016,L(99);:X$=INKEY$:IFX$
=""THEN110ELSED=ASC(X$):IFD>7ANDD<11
ORD=91THEN260ELSEGOSUB920:IFD<57ANDD
>48THEN230ELSEIFD<70ORD>90THEN110
106 OND-64GOTO110,110,110,110,110,29
0,110,110,110,110,110,110,110,110,11
0,110,240,110,250,210,110,110,110,11
0,110,110
110 IFL(104)=0THEN126ELSEX=L(101)+SG
N(L(97)-L(101))*L(105):Y=L(102)+SGN(
```

Fig. 13-2. Executive loop for Mazies and Crazies.

```
L(98)-L(102))*L(105):D=PEEK(X+64*Y+1
5360):IFD<>64THEN116
112 GOSUB920:PRINT@56,MID$(TB$,FIX((
L(104)-49)/6)*8+1,8);:PRINT@120,"ATT
ACKS!";:POKEL(97)+64*L(98)+15360,191
:POKEL(97)+64*L(98)+15360,64:L(105)=
-1:L(106)=RND(20):L(99)=L(99)-L(103)
/8:IFL(8)=91THENL(99)=L(99)+500
114 IFL(99)>0THEN122ELSE128
116 IFD=48THENFORI=1TO20:POKEX+64*Y+
15360,RND(64)+128:NEXT:POKEX+64*Y+15
360,32:L(16)=RND(87)+3:GOTO124
117 IFD=46THENPOKEX+64*Y+15360,32:L(
99)=L(99)+RND(4000)+2000:GOTO120:ELS
EIFD=43THENPOKEX+64*Y+15360,32:L(99)
=10000:L(11)=RND(87)+3:GOTO120
118 IFD=32THENPOKEL(101)+64*L(102)+1
5360,32:L(101)=X:L(102)=Y:POKEX+64*Y
+15360,42:GOTO120
119 IFL(105)=-1THENL(105)=1ELSEFORI=
1TO3:X=L(101)+RND(3)-2:Y=L(102)+RND(
3)-2:D=PEEK(X+64*Y+15360):IFD<>32THE
NNEXT:ELSE118
120 IFL(105)=-1THENL(106)=L(106)-1:I
FL(106)<=0THENL(105)=1
122 IFL(107)>1THEN126ELSEL(103)=L(10
3)-1:IFL(103)>0THEN126
124 POKEL(101)+64*L(102)+15360,32:PR
INT@56,"AT LAST!";:PRINT@120,"ITS DE
AD";:L(L(104))=0:L(104)=0
126 L(107)=L(107)-1:IFL(107)>0THEN10
4ELSEL(107)=10:L(99)=L(99)-1:IFL(99)
>0THEN104
128 FORI=1TO8:L(C(I))=L(0):C(I)=0:NE
XTI:L(100)=L(100)+1:CLS:PRINT@512,"Y
OU ARE QUITE DEAD. BUT WE CAN RESURR
ECT YOU!
WHEN YOU ARE READY, PRESS <ENTER>";
130 X$=INKEY$:IFX$=CHR$(13)THENCLS:G
OTO16:ELSE130
```

Fig. 13-2. Continued from page 207.

```
                HANDLERS

                 "TAKE"
              (Press T Key)

210 FORK=1TO8:IFC(K)<>0THENNEXTK:PRI
NT@56,"TOO MUCH";:PRINT@120,"TO CARR
Y";:GOTO110
212 N=L(97)+64*L(98)+15295:FORI=NTON
+128STEP64:FORJ=0TO2:A=PEEK(I+J)-32:
IFA>0ANDA<17ANDA<>10ANDA<>6THEN214EL
SENEXTJ,I:PRINT@56,"NOTHING";:PRINT@
120,"TO TAKE!";:GOTO110
214 POKEI+J,32:IFA=11THENL(99)=10000
:PRINT@56,"HEALTHY";:PRINT@120,"AGAI
N!";:L(11)=RND(87)+3:GOTO110:ELSEIFA
=14THENL(99)=L(99)+RND(4000)+2000:PR
INT@56,"THAT WAS";:PRINT@120,"TASTY!
";:GOTO110
216 IFA=4THENFORA=17TO48:IFL(0)<>L(A
)THENNEXTA:
218 L(A)=91:C(K)=A
220 PRINT@56,"OKAY!";:PRINT@120,"
     ";:GOTO102
```

Fig. 13-3. Handler Take.

```
                 "DROP"
           (Press Numeric Key)


230 N=L(97)+64*L(98)+15295:FORI=NTON
+128STEP64:FORJ=0TO2:IFPEEK(I+J)<>32
THENNEXTJ,I:PRINT@56,"NOWHERE";:PRIN
T@120,"TO DROP!";:GOTO110
232 K=C(D-48):L(K)=L(0):IFK>16THENK=
4
234 POKEI+J,K+32
236 FORI=D-48TO8:C(I)=C(I+1):NEXTI:G
OTO102
```

Fig. 13-4. Handler Drop.

```
                "QUIT"
            (Press Q Key)


240 J=0:FORI=17TO48:IFL(I)<>1THENNEX
T:ELSEJ=J+I:NEXT
242 FORI=49TO96:IFL(I)<>0THENNEXT:EL
SEJ=J+FIX((I-1)/6)+1:NEXT
244 J=J-L(100)*30:PRINT@184,"IF YOU
 ";:PRINT@248,"WERE TO ";:PRINT@312,
"STOP NOW";:PRINT@376,"YOUD    ";:PR
INT@440,"HAVE A  ";:PRINT@504,"SCORE
 OF";:PRINT@568,J;:PRINT@632,"WANT T
O ";:PRINT@696,"GIVE UP?";:PRINT@760
,"(Y OR N)";
246 X$=INKEY$:IFX$="N"THENFORI=0TO14
:PRINT@56+I*64,"        ";:NEXT:GOTO
102ELSEIFX$="Y"THENPRINT@0,"";:END:E
LSE246
```

Fig. 13-5. Handler Quit.

```
               "SHOOT"
            (Press S Key)


250 IFL(9)<>91THENPRINT@56,"YOU HAVE
";:PRINT@120,"NO BOW!";:GOTO110:ELSE
IFL(13)<>91THENPRINT@56,"YOU HAVE";:
PRINT@120,"NO ARROW";:GOTO110:ELSEIF
L(104)=0THENPRINT@56,"ZZINGG!!";:PRI
NT@120,"        ";:GOTO256
252 IFL(103)>5000THENPRINT@56,"BOUNC
ES";:PRINT@120,"OFF HIM!";:GOTO256:E
LSEN=SQR((L(101)-L(97))[2+(L(102)-L(
98))[2):N=81-N:IFRND(100)>NTHENPRINT
@56,"RATS!!";:PRINT@120,"MISSED!";:G
OTO256
254 L(13)=L(0):POKEL(101)+64*L(102)+
15360,45:A=13:GOSUB910:L(L(104))=0:L
(104)=0:PRINT@56,"GOT HIM!";:PRINT@1
20,"VICTORY!";:GOTO102
256 GOSUB940:L(13)=L(0):POKEX+64*Y+1
5360,45:A=13:GOSUB910:L(105)=-1:L(10
6)=RND(20):GOTO102
```

Fig. 13-6. Handler Shoot.

```
             "MOVE"
        (Press Arrow Key)
260 X=0:Y=0:IFD=91THEND=11
262 OND-7GOTO264,266,268,270
264 X=-1:GOTO272
266 X=1:GOTO272
268 Y=1:GOTO272
270 Y=-1:GOTO272
272 X=X+L(97):Y=Y+L(98):Q=PEEK(X+64*
Y+15360)
274 IFQ=32THENPOKEL(97)+64*L(98)+153
60,32:L(97)=X:L(98)=Y:POKEX+64*Y+153
60,64:L(99)=L(99)-5:IFL(99)>0THEN110
ELSE128
276 IFQ=48THENPRINT@56,"BOOMM!!";:PR
INT@120,"YOU FOOL";:FORI=1TO40:POKEX
+64*Y+15360,RND(64)+128:NEXTI:L(16)=
RND(87)+3:GOTO128
278 IFQ=35THENI=RND(87)+3:IFL(1)<>91
ANDL(1)<>ITHENGOSUB920:PRINT@56,"NOT
HING";:PRINT@120,"HAPPENS!";:GOTO110
:ELSEPRINT@X+64*Y,"POOF!";:FORI=1TO2
0:NEXTI:L(0)=I:GOTO100
280 IFQ=38THENPOKEL(97)+64*L(98)+153
60,32:L(97)=X:L(98)=Y:POKEX+64*Y+153
60,64:L(99)=L(99)-RND(200)+100:GOSUB
920:PRINT@56,"FIRE!!";:PRINT@120,"YE
OWW!!";:IFL(99)>0THEN110ELSE128
282 IFQ=128THENIFL(1)<>91THEN288ELSE
FORI=1TO91STEP10:IFX=VAL(MID$(CR$,I,
2))ANDY=VAL(MID$(CR$,I+2,2))THENL(0)
=VAL(MID$(CR$,I+4,2)):IFL(0)=92THEN2
86ELSEL(97)=VAL(MID$(CR$,I+6,2)):L(9
8)=VAL(MID$(CR$,I+8,2)):GOTO100:ELSE
NEXTI
284 GOTO110
286 PRINT@56,"OH NOOO!";:PRINT@120,"
A PIT!!";:FORI=1TO20:NEXT:GOTO128
288 GOSUB920:PRINT@56,"TOO DARK";:PR
INT@120,"IN THERE";:GOTO110
```

Fig. 13-7. Handler Move.

```
                  "FIGHT"
                (Press F Key)

290 IFL(15)<>91THENPRINT@56,"YOU HAV
E";:PRINT@120,"NO SWORD";:GOTO110:EL
SEIFL(104)=0THENPRINT@56,"FIGHTING";
:PRINT@120,"SHADOWS?";:GOTO298
292 IFABS(L(97)-L(101))>1ORABS(L(98)
-L(102))>1THENPRINT@56,"MISSED";:PRI
NT@120,"IT! FIE!";:GOTO298
294 L(103)=L(103)-L(99)/5:IFL(103)>0
THENPRINT@56,"A GOOD";:PRINT@120,"SL
ASH!";:IFRND(2)=2THENL(105)=-1:L(106
)=RND(20):GOTO298:ELSE298
296 PRINT@56,"FINISHED";:PRINT@120,"
HIM OFF!";:L(L(104))=0:L(104)=0:POKE
L(101)+64*L(102)+15360,32
298 L(99)=L(99)-L(99)/20:IFL(99)>0TH
EN110ELSE128
```

Fig. 13-8. Handler Fight.

```
                 SUBROUTINES

                  "DSPLAY"

500 L(104)=0:CR$=R(L(0)):PRINT@0,STR
ING$(56,191);:PRINT@960,STRING$(56,1
91);:FORI=64TO896STEP64:PRINT@I,CHR$
(191);STRING$(54,32);CHR$(191);:NEXT
I:N=1
502 IFMID$(CR$,N,1)="-"THENN=N+1:GOT
O504ELSEX=VAL(MID$(CR$,N,2)):Y=VAL(M
ID$(CR$,N+2,2)):POKEX+64*Y+15360,128
:N=N+10:GOTO502
504 IFN>LEN(CR$)THEN520ELSEX=VAL(MID
$(CR$,N+1,2)):Y=VAL(MID$(CR$,N+3,2))
:ONVAL(MID$(CR$,N,1))GOTO506,508,508
,508,510,512:GOTO520
506 PRINT@X+64*Y,STRING$(VAL(MID$(CR
```

Fig. 13-9. Subroutine Dsplay.

```
$,N+5,2)),191);:N=N+7:GOTO504
508 S=X+64*Y+15360:D=VAL(MID$(CR$,N,
1))+61:FORI=1TOVAL(MID$(CR$,N+5,2)):
POKES,191:S=S+D:NEXTI:N=N+7:GOTO504
510 S=X+64*Y-65:PRINT@S,"&&&";:S=S+6
4:PRINT@S,"&&&";:S=S+64:PRINT@S,"&&&
";:N=N+5:GOTO504
512 POKEX+64*Y+15360,46:N=N+5:GOTO50
4
520 POKEL(97)+64*L(98)+15360,64:FORI
=1TO16:IFL(I)=L(0)THENGOSUB940:POKEN
,I+32
522 NEXTI:FORI=17TO48:IFL(I)=L(0)THE
NGOSUB940:POKEN,36
524 NEXTI:FORI=49TO96:IFL(I)=L(0)THE
NGOSUB940:POKEN,42:L(101)=X:L(102)=Y
:L(104)=I:ELSENEXTI
526 RETURN
```

Fig. 13-9. Continued from page 212.

```
                "STATUS"

900 PRINT@248,"ROOM";L(0);:PRINT@952
,"STRENGTH";:PRINT@1016,L(99);
902 FORI=1TO8:IFC(I)=0THENPRINT@312+
I*64,"          ";:RETURN:ELSEPRINT@31
2+I*64,CHR$(I+48);" ";:IFC(I)>16THEN
PRINT"TR";C(I)-16;:NEXT:RETURN:ELSEP
RINTMID$(TA$,C(I)*6-5,6);:NEXT:RETUR
N
```

Fig. 13-10. Subroutine Status.

```
                "SUBINV"

910 FORI=1TO8:IFC(I)<>ATHENNEXTI:RET
URN:ELSEFORJ=ITO8:C(J)=C(J+1):NEXTJ:
RETURN
```

Fig. 13-11. Subroutine Subinv.

```
                 "CLRMES"

920 PRINT@56,"              ";:PRINT@120,"
         ";:RETURN
```

Fig. 13-12. Subroutine Clrmes.

```
                 "RANDXY"

940 X=RND(54):Y=RND(14):N=X+64*Y+153
60:IFPEEK(N)<>32THEN940ELSERETURN
```

Fig. 13-13. Subroutine Randxy.

```
              ROOM STRINGS

1000 R(1)="0006025408550703010629009
02714-4010111254011111180418"
1001 R(2)="5508010106190000423140009 0
55408201506250l-3060112312031211307 3
9"
1002 R(3)="0006015407211507270155070
8010922000929l4-1011027327040712804 2
2"
1003 R(4)="2315021901240010311401001
14014-3040108105084531909 04"
1004 R(5)="5508020109000212541100121
35403-1010850351040950202"
1005 R(6)="2500022014011514410126151
53201-3050609106062733107 04"
1006 R(7)="2700032114281516330139151
71401-104045 1"
1007 R(8)="0009035407551018010555021
90110-3200105324031255310"
1008 R(9)="2915032201400020151430002
13414-2360112436011 2"
1009 R(10)="31150424010006225406-331
0408127090 9"
```

Fig. 13-14. Room strings.

```
1010 R(11)="401504010110002244l4-108
05382460510"
1011 R(12)="55110501020012235402-101
10203241005"
1012 R(13)="55030501120004235412-421
01062330106"
1013 R(14)="41000601141315244501-101
1230330021 2"
1014 R(15)="32000626140007245407-101
0538"
1015 R(16)="33000728145508250108-345
01081210928"
1016 R(17)="14000739144215251601-101
04204010507"
1017 R(18)="00050854105501260113-317
01103210411325011 0"
1018 R(19)="00100854025511260103-104
09431200520"
1019 R(20)="15150940014300271714-442
01091200931"
1020 R(21)="34150930015509270109-108
05291081029336051 0"
1021 R(22)="550610010644151110010315
285404-22806084280609"
1022 R(23)="550212011255121301040008
285408-10105351011035336050 4"
1023 R(24)="450014131455071501070900
285414-2540110244011 0"
1024 R(25)="000816540816001742145000
290110-34901051030545"
1025 R(26)="001318540100031954115508
290107-3150107315090634501 10"
1026 R(27)="171520430100092154095215
290105-3180411"
1027 R(28)="550422031455082301085514
2409014615300301-450020525409051030 6
4460510"
1028 R(29)="001025500100072654080005
2752140415305001-401060710812381460 4
```

Fig. 13-14. Continued from page 214.

```
0961804"
1029 R(30)="0300284614500029041427l5
312901-328011010605 42"
1030 R(31)="000632540855073301062900
302714-401011125401111180418"
1031 R(32)="550831010619003423140009
355408201536250l-306011231203121l307
39"
1032 R(33)="000631540721153727015507
380109220039 2914-101102732704071280 4
22"
1033 R(34)="231532190124004031140100
414014-304010810508453190904"
1034 R(35)="550832010900024254110012
435403-101085035104095020 2"
1035 R(36)="250032201401154441012615
453201-305060910606273310704"
1036 R(37)="270033211428154633013915
471401-1040451"
1037 R(38)="000933540755104801055502
490110-3200105324031255310"
1038 R(39)="291533220140005015143000
513414-2360112436011 2"
1039 R(40)="31153424010006525406-331
04081270909"
1040 R(41)="40153401011000524414-108
0538246051 0"
1041 R(42)="55113501020012535402-101
1020324100 5"
1042 R(43)="55033501120004535412-421
0106233010 6"
1043 R(44)="41003601141315544501-101
1230330021 2"
1044 R(45)="32003626140007545407-101
0538"
1045 R(46)="33003728145508550108-345
0108121092 8"
1046 R(47)="14003739144215551601-101
0420401050 7"
```

Fig.13-14. Continued from page 215.

```
1047 R(48)="000533541055015601l3-317
01103210411325011O"
1048 R(49)="0010385402551156O103-104
09431200520"
1049 R(50)="15153940014300571714-442
01091200931"
1050 R(51)="34155930015509570109-108
0529108102933600510"
1051 R(52)="5506400106441541100103l5
585404-22806084280609"
1052 R(53)="5502420112551243010400O8
585408-1010535101103533605O4"
1053 R(54)="450044131455074501070900
585414-254011024401l0"
1054 R(55)="000846540816000474214500O
590110-3490105103054 5"
1055 R(56)="001348540100034954115508
590107-315010731509063450110"
1056 R(57)="171550430100095154095215
590105-3180411"
1057 R(58)="550452031455085301085514
540901461560O301-450020525409051030 6
4460510"
1058 R(59)="001055500100075654080005
575214041560 5001-401060710812381460 4
0961804"
1059 R(60)="030058461450005904142715
612901-3280110106054 2"
1060 R(61)="000662540855076301062900
602714-40101112540111118O418"
1061 R(62)="550861010619006423140009
655408201566250 1-306011231203121130 7
39"
1062 R(63)="000661540721156727015507
680109220069291 4-101102732704071280 4
22"
1063 R(64)="231562190124007031140100
714014-3040108105084531909 04"
1064 R(65)="550862010900027254110012
```

```
735403-101085035104095O202"
1065 R(66)="250062201401157441012615
753201-305060910606273310704"
1066 R(67)="270063211428157633013915
771401-1040451"
1067 R(68)="000963540755107801055502
790110-3200105324031255310"
1068 R(69)="291563220140008015143000
813414-2360112436O112"
1069 R(70)="31156424010006825406-331
04081270909"
1070 R(71)="40156401011000824414-108
0538246O510"
1071 R(72)="55116501020012835402-101
10203241005"
1072 R(73)="55036501120004835412-421
01062330106"
1073 R(74)="41006601141315844501-101
12303300212"
1074 R(75)="32006626140007845407-101
0538"
1075 R(76)="33006728145508850108-345
01081210928"
1076 R(77)="14006739144215851601-101
04204010507"
1077 R(78)="00056854105501860113-317
011032104113250110"
1078 R(79)="00106854025511860103-104
09431200520"
1079 R(80)="15156940014300871714-442
01091200931"
1080 R(81)="34156930015509870109-108
052910810293360510"
1081 R(82)="550670010644157110010315
885404-22806084280609"
1082 R(83)="550272011255127301040008
885408-101053510110353360504"
1083 R(84)="450074131455077501070900
885414-25401102440110"
```

Fig. 13-14. Continued from page 217.

```
1084 R(85)="00087654081600077421450000
890110-34901051030545"
1085 R(86)="00137854010003795411550008
890107-31501073150906345011O"
1086 R(87)="1715804301000098154095215
890105-3180411"
1087 R(88)="550482031455088301085514
84090146159003O1-45002052540905103066
4460510"
1088 R(89)="0010855001000786540800005
8752140415905001-401060710812381460004
0961804"
1089 R(90)="0300884614500089041427155
012901-32801101060542"
1090 RETURN
```

Fig. 13-14. Continued from page 218.

Summary

How does anyone manage to summarize an entire book? I suppose the best way is to reiterate the key concepts, seeking to clarify things for some readers at the risk of boring others.

Let all who read heed concept one: structure your programming. No program of any real complexity can avoid the debug-correction cycle, and an unstructured program is difficult to correct. Some of the problems themselves may be linked to lack of structure, such as accidental re-use of variables best left untouched. One hour to make a flow chart or table on paper is worth the ten hours of confusion avoided at the keyboard later.

The second concept is even more crucial: consider your options. In an input-oriented program like Basements and Beasties, someone is bound to phrase a command or try something in a way you hadn't predicted—possibly with disastrous results to your program flow! Do a lot of if-then thinking and testing before presenting your adventure program to a prospective adventurer.

The final concept may take awhile: optimize your code. Find ways to simplify your statements and speed up program execution. Use calculated jumps (ON-GOTO) and subroutine calls (GOSUB). Study anything you can read on the way BASIC actually works, and find ways to manipulate it using PEEK and POKE. If you think "adventure programming" has reached its zenith, surprise, there are plenty of more tricks to be tried. You'll probably be the first to find some of them.

After all, if you're ready to fight dragons with the Axe, then maybe you're adventurous enough to be a programmer!

Index

Writing BASIC
Adventure Programs for the TRS-80

by Frank DaCosta

Find out how you can write your own original adventure programs for your TRS-80 Model I or III with as little as 16K of memory! Discover how to use new programming tricks and techniques to gain memory space and increase programming speed! Even learn how to construct a full-feature *graphic* adventure! It's all here in this unique programming guide, along with two brand new games devised by the author to help you perfect your game-writing skills!

The first step is to find out what an adventure program contains and how it is created. You'll learn to map a "basement" or scenario for your adventure, including all the elements the program needs to support—from writing the description of each room and its contents to constructing a complex map containing surprises for the unwary player. Structuring the program is covered and you'll learn special techniques for organizing the BASIC code to speed up data access and reduce memory usage.

You'll examine a sample adventure program called "Basements and Beasties" to find out how programs are initialized, how scenes are described, and how commands are input and executed. Motion commands, obstacles, the use of magic words and action routines become clear so that you'll be able to use them properly when you write your own original game.

The how-to's for constructing a graphic adventure including the concepts, the semgnets, and listings are given along with a sample graphics adventure, "Mazies and Crazies." If you're interested in computer games and want to learn how to write your own *and* improve your programming skills at the same time, then this book is definitely for you!

Frank DaCosta is a computerist who is experienced in both hardware design and software development. A computer hobbyist, he works professionally with many types of microprocessor- and minicomputer-based systems.

OTHER POPULAR TAB BOOKS OF INTEREST

30 Computer Programs for the Homeowner, in BASIC (No. 1380—$9.95 paper; $18.95 hard)
33 Challenging Computer Games for TRS-80™/Apple™/PET® (No. 1275—$7.95 paper; $14.95 hard)
67 Ready-To-Run Programs in BASIC: graphics, home & business, education, games (No. 1195—$7.95 paper; $13.95 hard)
The BASIC Cookbook (No. 1055—$5.95 paper; $10.95 hard)

TAB **TAB BOOKS Inc.**
Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

Prices higher in Canada

ISBN 0-8306-1422-2