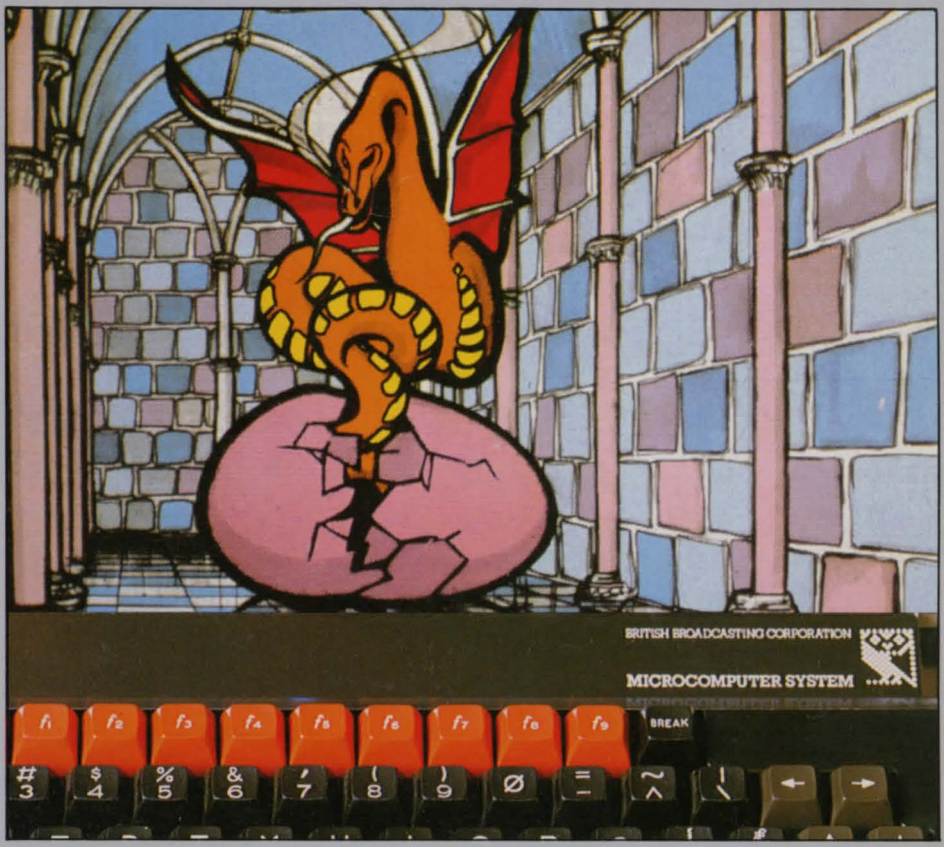
How To Write
ADVENTURE GAMES

for the BBC Microcomputer Model B and Acorn Electron

Peter Killworth



PENGUIN ACORN COMPUTER LIBRARY

Penguin Books
How To Write Adventure Games

Peter Killworth is the author of some of the best selling adventure games published by Acornsoft Ltd. for use on the BBC Microcomputer Model B and Acorn Electron. By profession he is a senior research associate at the Department of Applied Mathematics and Theoretical Physics in the University of Cambridge, and works in the field of dynamical oceanography and climate. Peter Killworth's current projects include a magic program with Paul Daniels and a mathematics adventure program designed to teach 13 year-olds who are studying the SMP syllabus.

Peter Killworth has also prepared a cassette program which can be used in conjunction with this book. It is available (price £9.95) from Acornsoft Limited, c/o Vector Marketing Ltd, Denington Industrial Estate, Wellingborough, Northants NN8 2RL. Please allow 28 days for delivery.

The object of this book is to enable a reader fairly fluent in standard BBC BASIC to create and write fairly complicated Adventure games without recourse to machine code; he or she may well learn some programming techniques as well! No previous knowledge is assumed. The ratio of text to program is very high for a book of this type: the intention is not to teach an intimate knowledge of three Adventure games to the exclusion of all others. Rather, it is to foster an understanding of how to create other programs and so allow the reader to move beyond the confines of what is explicitly covered in this book. To this end, both a database creation program and a 'shell' for an advanced Adventure game are provided.

How To Write ADVENTURE GAMES

Peter Killworth

Penguin Books

This book owes its existence to many people, and I thank them all: a student on a flight from Portland, Oregon to San Diego who first told me about Adventure and the joys of mazes and rod-waving; an anonymous writer in the Cambridge IBM 370 complaints book who grumbled how he couldn't find a free terminal as everyone was busy running M. Oakley's compilation of Colossal Cave; M. Oakley himself for providing it; David Seal and Jon Thackray for writing the first half of Acheton (still the best game!) and for teaching me what a good database system looked like; Jonathan Mestel, Ian Farquarson, and David Jeffrey for discussion and argument about Adventures both specific and general, and so many suggestions; Jonathan Partington for many enjoyable and puzzling hours; the people at Acornsoft, specifically first Chris Jordan and then Tim Dobson, and especially David Johnson-Davies, for their support and making this possible; Acorn USA for loaning much-needed equipment during summer 1983; Andrew for telling me what to do with the mushroom; Paul for creating the old lady one afternoon and donating much of the plot for the Roman adventure in this book; Addison-Wesley, for permission to use material which first appeared in Acorn User; Jonathan Griffiths and Rob Macmillan, who devotedly forced my programs into a vague semblance of structure; the writers of the Acornsoft View word processor, on which this book was written; and last, but really first, Sarah for putting up with seeing only a bent back, no husband, and all that clacking. . .

The Penguin Acorn Computer Library is a joint venture, produced by Acornsoft Limited (in association with Pilot Productions Limited), and published by Penguin Books Limited

Penguin Books Ltd, Harmondsworth, Middlesex, England
Penguin Books, 40 West 23rd Street, New York, New York, 10010, U.S.A.
Penguin Books Australia Ltd, Ringwood, Victoria, Australia
Penguin Books Canada Ltd, 2801 John Street, Markham, Ontario, Canada L3R 1B4
Penguin Books (N.Z.) Ltd, 182-190 Wairau Road, Auckland 10, New Zealand

First published 1984

Set in Palatino by Saxon Press, Norman House, Heritage Gate, Derby

Colour origination by RCS Graphics Ltd, 39-40 Springfield Mills, Farsley, Pudsey, Leeds, and MRM Graphics, 61 Station Road, Winslow, Bucks

Made and printed in Spain by Printer industria grafica s.a., Sant Vicenc dels Horts, Barcelona
D.L.B. 15171-1984

Line illustrations by Rob Shone
Original photography by Nick Wright

CONTENTS

INTRODUCTION

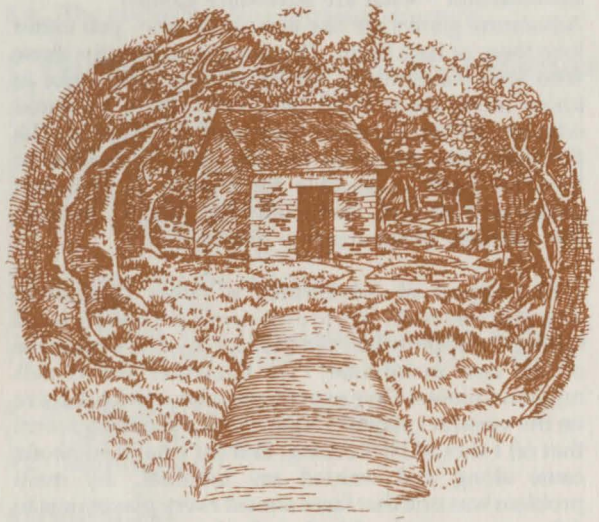
Introduction – what are adventure games?
Adventure games are like avocado pears – you either love them or hate them. This book is written for those who love Adventures, for those who would like to know more about them, and particularly for those who would like to create their own Adventures. The flexibility and power of the BBC Microcomputer or Electron make it possible for anyone to construct games of their own, using only BBC BASIC, without the necessity of lapsing into machine code.

The appeal of Adventures is rather difficult to describe; one really has to experience it. I vividly recall typing the appropriate symbols for my first ever game on the local mainframe. (The Crowther and Woods original Colossal Cave extended to about twice the size on the version I played.) Apart from a sneaking worry that (a) I should be working, and (b) what if someone came along and wanted my terminal, my main problem was one that faces almost every player new to Adventure games – I hadn't the faintest idea what I was supposed to do!

The game came up with 'Welcome to Adventure!', which was friendly, followed by 'Would you like instructions?' A slight problem there, I thought.

Obviously I did want instructions – but should I type 'YES' or 'Y' or 'yes' or 'y' in reply. It turned out not to matter – but it taught me later that it shouldn't matter to my players either if I could help it. The program obligingly told me that the computer was my eyes and hands. It would describe what I could see, and tell me what happened to me as a result of my actions. In return I must tell it what to do on my behalf, using sentences of one or two words. A few helpful examples were given: GET, DROP, INVENTORY, LOOK, HELP. I was told that only five letters were scanned, so that 'northeast' had to be typed as 'ne' to distinguish it from 'north'. The object, it transpired, was to find all the treasure and dump it in the house. Then I should re-enter the cave, when the 'end-game', whatever that was, would befall me. I remember assuming that none of this would take too long...

The description of where I was then appeared. 'You are standing at the end of a long road. In front of you is a brick building. A stream flows out of the building and down a valley. Around you is forest.'



And that was that. I was on my own, and utterly baffled. This feeling of bafflement seems to divide those who love Adventures from those who dislike them; the former treat it as a challenge, while the rest put the game away and do something else instead!

Tentatively I typed 'building' (I'd forgotten about the five letters already) to be met with 'You are standing inside the building...' and some more description. Obviously I had moved inside the building, which was what I had intended. What was even better was a pile of useful things on the floor – things like a lamp, a bottle of water, some food, and so on. I tried 'eat food' to be met by 'You're not holding that!' which was frustrating; why couldn't the silly machine let me bend down and scoff the food? O.K., so 'take food' was followed by 'OK', to which I said 'eat food'. The response 'Thank you, that was delicious!' worried me. It looked as if that food had some other use than feeding my alter ego's stomach. (Stirrings of the Adventure player's first maxim: almost everything has a use, otherwise the writer would not have written it into the game!)

Some little time later (well, about an hour actually) I made a save of my position (I couldn't get past a pesky snake, an obstacle which will be familiar to many readers) and headed home in great excitement. The game was tremendous!

Perhaps the easiest way to show the appeal of Adventures is to give you the to-and-fro conversation between player and machine in an actual game. This particular one is called, ''Countdown to Doom''. It has a science fiction flavour: the player has crashed his spaceship on the planet Doom and has 220 turns to repair it before the ship corrodes. Now read on (player responses are in capitals for clarity).

You are standing in what used to be the control room of your spaceship, but after the attack it looks more like a junkyard. The only usable exit is northwards, as the other exits are covered by a delicate patina of squashed solid-state circuitry. On the floor there is some explosive with a self-igniting fuse. A ruined console lies to one side, barely operating. Only one button seems to be functional...

PRESS BUTTON Nothing happens

GET BUTTON You can't take that!

GET EXPLOSIVE OK

GO NORTH You are in what used to be the engine

room of your ship, only in the landing the cargo hold has been concertinaed into it. What's left of the motor will not, alas, be responsible for any more flights. There is a port south, and a doorway northeast. The door to the northeast was jammed on landing and won't open

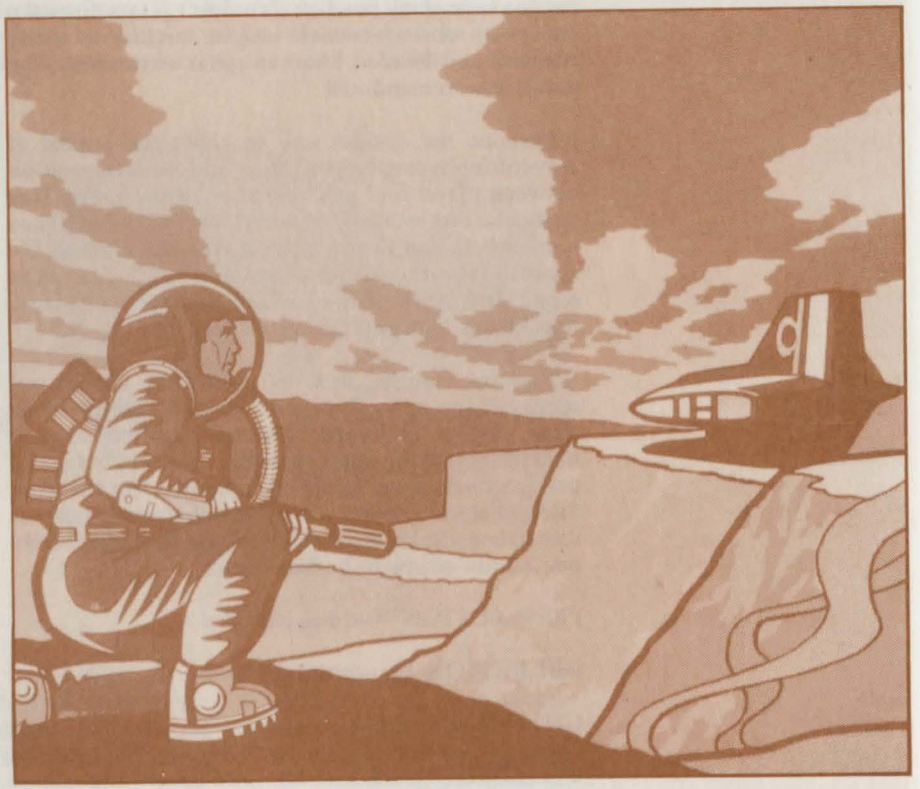
NE The door to the northeast was jammed on landing and won't open

OPEN DOOR The door to the northeast was jammed on landing and won't open

THROW EXPLOSIVE OK

GET OK

LIGHT FUSE The fuse starts to burn Time before ship collapse is 210.



DROP EXPLOSIVE OK

S You're in the control room A ruined console lies to one side, barely operating. Only one button seems to be functional

WAIT From next door there is a loud bang! OK

BACK You're in the cargo hold-cum-engine room. The door to the northeast is off its hinges but still blocks the way

NE The door to the northeast is off its hinges but still blocks the way

PUSH DOOR The door falls outwards onto the ground, leaving the way clear

NE You are standing under a dull copper sun in a wide open clearing where your ship crashed. To the northeast, north and northwest lies an evil-looking (and smelling!) swamp, while further west a track climbs steeply up the lower tongue of a glacier. Westwards there seems to be jungle, while south an opening in the mountains reveals a valley. Southeast there is a path into the mountains, and a narrow path wanders east through cliffs to skirt round the swamp. To the southwest lies your ship. There is a heavy mangled door here

GET DOOR You can't take that!

E You're at a flat area of scorched ground. Cliffs to the west fall away to the swamp. Paths lead northeast, east, and southwest. A metallic object is south. A wide hole has been scooped out of the earth. There is a small jelly-like blob here. It is wriggling towards the cliffs

GET BLOB As you touch the blob, a thousand volts pass through you. Shocking, isn't it? You seem to have lost your life. You have scored 0 out of 250 Would you like another game?

Analysing my reactions years after playing my first game, I see them as precisely those of avid Dungeons and Dragons or Traveller or other role-playing gamers. I wasn't just playing Adventure, and trying to solve the problems that came my way by native wit/common sense; I was living the game. The

machine was providing me with excellent descriptions of areas, objects, disasters, etc. (plus some very funny jokes at my expense), but I could see the caves I was exploring in my imagination. My maps, though scrappy, became treasured friends; each new problem surmounted (or cryptic hint from a friend decoded) became a source of soaring emotion. In short, I was hooked, and still am, many, many games later.

Adventure games vary both in style and background. Some, like Zork (available on some micros and various mainframes) allow vast sentences in English like 'take all but the rope and throw the pig at the cat'. Some provide pretty pictures of various scenes. Some disc-based American products even include moving graphics, and a lot of disc-whirring, no doubt! Some even talk to you in computer-speak. Some are hilariously funny; some are fiendishly difficult even for experts. Some suffer from crude programming or lots of errors.

This book is about how to write games like these (but hopefully not the ones with lots of errors). It's not totally about programming, however, it's also about the Adventure plots; about user-friendliness, whatever that means; and, at the end of it all, it is about having fun. Because that's what Adventures are for.

Before we begin the first part of this book, you must learn the fundamental rule of Adventure programming:

No matter how small an Adventure you write, it will take far, far more time and effort than you thought it would.

A map of this book

This book is divided into eight Parts, most of which are a mixture of discussion and programming.

Part 1 talks about the requirements of Adventure games, and how they might be implemented on a microcomputer.

Part 2 creates a simple, menu-driven game of the 'hack-and-slash', exploring variety; it was chosen not because this is a good Adventure game model, but because we can learn some useful concepts from it.

Part 3 designs a simple Adventure, called 'MINI' because it only has four rooms. Nonetheless it contains four puzzles, and introduces many ideas which will recur later. You can use the game system it employs, without ever using the more complex system described later.

Parts 4 to 7 are devoted to the creation and programming of an advanced Adventure set in ancient Rome.

Part 4 develops the plot, and refines and completes the database system for use in such games.

The price paid for a flexible system is the need for a separate program to insert the data into the computer, and Part 5 is devoted to such a program, which can of course be used for any game.

Part 6 details the subprograms that allow the database to be manipulated by the Adventure program, and the overall running structure. In other words, Part 6 creates a 'shell' for use in any Adventure.

Part 7 finally programs the 'ROMAN' game proper, and includes details as to how to dovetail the program and database.

Part 8 is in many ways the most important, although it contains no programming details. It is largely devoted to a discussion about plot creation and implementation.

Two appendices are provided which give very brief looks at bitwise logic and hexadecimal notation. Neither are intended as full treatises on the subject!

HOW ARE GAMES WRITTEN?

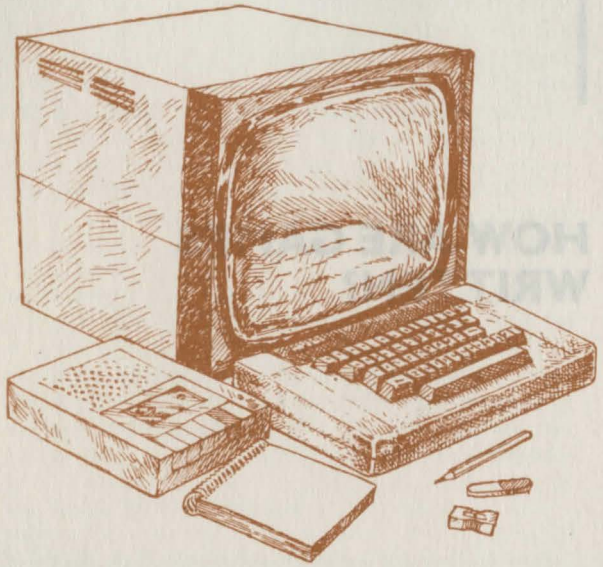
1.1 Fundamentals

If we're going to write some games, we need the right equipment. The BBC Model B or Electron, either a black and white or colour TV or monitor, and a cassette recorder are all the hardware required. You don't need a disc pack, although that would speed things up. The only other essential hardware is stationery: pencils (sharp), an eraser, and LOTS of paper.

Most of what we need is mental. Obviously an ability to write in BBC BASIC is necessary, though much of this can be learned as we go along. A little facility in mathematics will help, but isn't vital. A good imagination is, however, as is the ability to keep calm and organized.

So what ingredients are important in Adventure games? Let's step away from the programming problems for the moment and think from the player's point of view. We have to supply:

a) a PLOT (maybe weak, maybe strong; but the player must have a goal or else why play the game?)
b) an AREA, or a GEOGRAPHY (usually to explore and map, but not necessarily)



c) some TOOLS for the game (usually objects or information which the player manipulates to achieve his goals)
d) a way for the player to make decisions and TELL them to the game (or INPUT them)
e) a way to TELL THE PLAYER WHAT HAPPENS.

On top of this, there are some other features which aren't so essential; for example, the ability to save a game in the middle rather than always having to start at the beginning. We'll ignore these finer points until we start constructing more complicated examples.

Although (a) to (e) above are what the player sees, it's important to realise that they are not necessarily what the writer sees. Indeed, the programs must concentrate on far more:

f) any game must have a well-defined VOCABULARY (whether the player knows this from the start or discovers it during the course of the game, depends on you)
g) a way of handling ERRORS of all kinds, from unknown words to inconsiderate pressing of 'Escape' at awkward moments
h) a way to keep track of all the things going on,

where the player is, what happened to that dragon, and so on; this is known as a DATABASE structure
i) an overall RUNNING PROGRAM to organise all the strange things that happen to the player.

Finally, there are one or two other items which should be mentioned right now. One is the question of STORE, the other concerns GRAPHICS.

I stated earlier that you will need a Model B or Electron, but in fact our first two games will run on a Model A. Quite frankly 32K, which sounded awfully large to me when I started, shrinks rapidly when dealing with real programs. Because of this, don't consider graphics unless you have access to a second processor. It can be done at the cost of grossly simplifying the game, but the real drawback is that even switching from the 'cheapest' mode – 7 on the BBC Micro, 6 on the Electron – to a graphics mode like 4 can lose 9216 bytes of storage. Think of a byte as a character, if the word bothers you, and just take it as read that we can't afford to lose that much store normally.

Let's now look at some of the Adventure ingredients which we mentioned above.

1.2 Vocabulary

The player of an Adventure will need a vocabulary with which to communicate. Frequently, in other sorts of program, users have to instruct a program to run. Typically, the user is presented with a MENU (i.e. a list) of alternative inputs, each with their own meaning. For example, a menu might contain the following instructions: 'Type 1 to reset the colours'; 'Type 2 to rescale the graph'; 'Type 3 to see the graph from a different angle'; and so on. A menu of this or a similar kind is convenient for most programs because it pre-plans both the sense and format of the user's response, and ensures that the computer will understand.

In an Adventure, where discovering the vocabulary of communication is part of the game, this method is obviously inappropriate. The program will not ask a player to enter '3' when he wishes to throw a rock. It will be up to the player to say 'throw rock' or 'hurl stone' or something of that sort.

This immediately gives us two problems. First there is the thorny one of SYNONYMS. The player will get a bit cross if his favourite verb (like 'hurl' in the example) is met by a bland 'I don't understand that!' But the poor programmer is in a no-win situation. If he includes 'hurl' in the computer's vocabulary to satisfy one player he may alienate another because 'sling' isn't understood. . . There is no easy solution here; you have to decide just how much vocabulary you want to teach the program, and then call a halt.

The other problem with constructions such as 'throw rock' is getting the program to understand it even when the vocabulary has been included! To begin with, the player may leave, say, three spaces between 'throw' and 'rock' when you expected a mere one. Or he may just say 'throw' and the computer expects an object after the verb. Or, even worse, what if he says 'rock' without the 'throw' and your program is expecting a verb, not a noun?

The obvious solution, and one which works very well for small vocabularies, is to make a list of all the verbs your program will accept and store them somewhere with easy access. One place – but not necessarily the best, as we'll see – would be in a subscripted string array.

Why an array? (I assume the 'string' part is obvious – where else could you store characters?) Well, you need to be able to do two quite different things with the player's verb. First, you have to check to see if you recognise it. Second, you need to take some action based on what verb it was.

Now you could do this very simply with a piece of program like this (ignoring nouns for now):

```
 80 dead=FALSE: REM use dead as a variable to end
 90 REPEAT                                  game with
100 INPUT "What now?" X$
110 IF X$="GET" PROCGET:GOTO 200
120 IF X$="DROP" PROCDROP:GOTO 200
130 IF X$="THROW" PROCTHROW:GOTO 200
140 ... etc...
...
200 UNTIL dead
```


which, as I said, is fine for a very small vocabulary but becomes tedious – and space-consuming – for maybe 60 commands. So let's reorganise how we can store the vocabulary by putting it in an array VOCAB$(I%), where I% runs from 1 to 60. Then we could write something like:

```
 10 DIM VOCAB$(60)
 20 FOR I% = 1 TO 60: READ VOCAB$(I%): NEXT
...
 80 dead=FALSE
 90 REPEAT
100 INPUT "What now?" X$
110 I%=0
120 REPEAT I%=I%+1
130 UNTIL X$=VOCAB$(I%) OR I%>59
140 IF X$<>VOCAB$(I%) PRINT "EH??":UNTIL FALSE
150 ON I% GOSUB 2000,2100,.....
...
200 UNTIL dead
```

The idea here is that we tuck all the vocabulary away in a DATA statement somewhere towards the end of the program; something like

```
5000 DATA GET,DROP,THROW,EAT,...
```

and use line 20 to store these words in the array VOCAB$. After getting X$ read in (line 100) we repeat-loop through I% in lines 120 and 130 until we find the appropriate element of VOCAB$ which matches X$. If we never do find a match (line 140) we grumble and let the player try again. Otherwise (line 150) we know which word it was because we know I%, and can act on it for example as shown in line 150.(I don't like nasty anonymous GOSUBs, but they are very useful in cases like that, although there are other methods.)

The astute reader will be asking various questions – or at least I hope you are. First, why use a clumsy REPEAT loop when a FOR loop on I% would do just as well? Because when we found the matching I%, we'd have to jump out of the loop somehow. BBC BASIC sensibly frowns on such activities; frankly, any program trying that deserves what it gets. Second, why I% rather than I? Because integer variables –

those with % signs after them – perform much faster than real variables; and 'resident' integer variables (A% to Z%) are a bit faster still. If you've never tried it, type in the following:

```
10 TIME=0
20 FOR I = 1 TO 10000
30 NEXT
40 PRINT TIME
```

and then repeat with I changed to I%. The saving – nearly a factor of three – is dramatic! Since Adventure games often do quite complicated things between commands, speed is essential; and that usually means employing integer variables.

Third, you might be wondering why the vocabulary was stored so wastefully. After all, that DIM statement took up quite a bit of store (256 bytes, that's 1/4 of 1K) and we ended up storing the vocabulary twice: once in the program in the DATA statement, and once in the VOCAB$ array. There are ways round this waste as we'll see later. We could leave the vocabulary in the DATA statement, and each time RESTORE to the beginning of the DATA line, reading in until we find a match:

```
 90 REPEAT
100 INPUT "What now?" X$
110 RESTORE 5000
120 I%=0
130 REPEAT
140 I%=I%+1
150 READ Y$
160 UNTIL X$=Y$ OR I%>60
170 IF X$<>Y$ PRINT "EH??": UNTIL FALSE
180 ON I% GOSUB...
```

This is slightly slower than using a subscripted array, but not much. In either case, time is spent in the process of checking. In the end, I prefer the original method because it's neater. Neat programs tend to demand less debugging time, and adding some extra information with each piece of vocabulary, such as whether to expect another word, you'll be reading that in as well; all extra time.

So part of our programming is going to involve a playoff between SPACE and TIME. A DATA list or VOCAB$ array is just too slow to sort through a fair size vocabulary. And, later on, we'll see how the concept of alphabetizing (a useful sorting technique) will speed up the hunt rather dramatically.

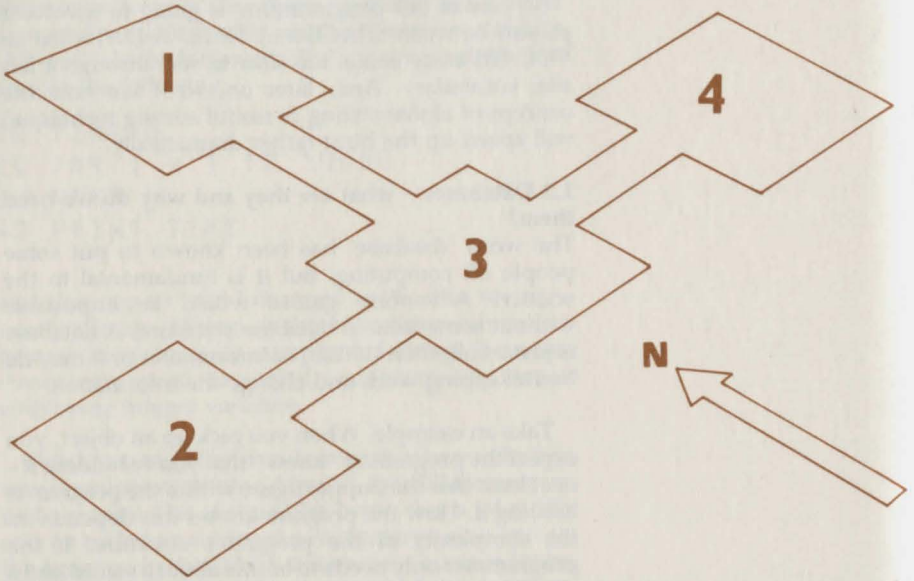
1.3 Databases – what are they and why do we need them?

The word 'database' has been known to put some people off computing, but it is fundamental to the science. Adventure games would be impossible without some form of database structure. A database is just a collection (or file) of information or it may do housekeeping work and change the information.

Take an example. When you pick up an object, you expect the program to 'know' that you're holding it – or at least that the puppet figure within the program is holding it. How the program knows this depends on the complexity of the program's structure. If the programmer only needs to bother about a sword and a shield (carrying these is the sole condition of a player being able to enter a room containing a fire-eating dragon safely, say), then we could just have two variables SWORD and SHIELD set to FALSE originally, and TRUE after the player had picked them up. (I prefer using logical values to SWORD=1 when the player is carrying it for two reasons: it's quicker to test IF SWORD THEN . . . than IF SWORD=1 THEN . . . , and I can more easily remember what the variable means!)

That simple example is a database, albeit a trivial one. It's a set of accessible variables – in this case exactly two – that the program can examine at any time to calculate the player's situation. Most adventure databases are complicated by the need for areas to be explored and mapped. The computer's-eye-view of such a map is a database too, since the program has to know whether the player can, for example, go west from his current position (or whether a sack of concrete will fall on him, or whatever. . .). So how could we store such a map?

Many people find drawing maps difficult enough, let alone converting them to numerical terms which a computer can understand! Let's look at a simple example. Let the map have four areas, or rooms, only:



North is up the page. Room 1 has a single exit south (to 3): room 2, a single exit east (to 3); room 3 has exits west, north and east (to 2,1 and 4), and room 4 has a single exit west (to 3). Each exit is also an entrance in the example, but this need not always be true (e.g. when an exit leads to a trap or cliff). As north, west, etc. are real directions, to return to a room you must travel in precisely the opposite direction. Again, this is not necessarily true since twisty passages, mazes, etc. can complicate matters.

The names of the rooms above are fairly boring, of course. They could have been 'A glade in the forest', 'The giant's castle', and so on. The player will certainly prefer it that way! So, why number the rooms? That's important. Computers are much happier working with numbers than with names – it's easier for the program to ask 'is the player in room 1?' and to look at some variable then it is for it to ask 'is the player in the forest glade?' It means that somewhere in the innards of the program there'll have to be a piece of text reading 'A glade in the forest' and some means of relating this to room 1. But the label of the room apart, we'll have to resign ourselves to thinking of that glade by a prosaic number.

How do we tell the program about the layout of the four rooms in the diagram? There are literally dozens of ways of doing it, ranging from very simple to very complex. How cunning a method you use will depend on how cunning the problems are in your scenario, together with how much space you have left over for your method.

At the very least, when specifying a room number, the database must allow the program immediate access to: (i) the directions of the exits, and (ii) the destinations of those exits. We could store this information in several ways.

We could set up a series of dimensioned arrays NORTH(4), EAST(4), SOUTH(4), and WEST(4). The entry in EAST(2) would refer to your destination (room) when going EAST from room 2 (i.e. room 3). This is fine, but what of NORTH(1)? There is no exit north from room 1. However, we must inform the program of this. Zero will do fine, provided that there is no room with that number .

If we follow this method of holding the map, we shall have entries looking like this:

ARRAY	INDEX	1	2	3	4
NORTH		0	0	1	0
EAST		0	3	4	0
SOUTH		3	0	0	0
WEST		0	0	2	3

This isn't too bad. If the player wants to go SOUTH, say, all we have to do is check SOUTH(X) (where X signifies the room he's in). If it's zero, we politely tell the player he can't go that way, and let him do something else. If it's non-zero, then we set X=SOUTH(X). This resets the player's room to his destination. We'll probably have to tell him what the new room looks like, and will deal with that later.

This method, albeit easy to understand, has a few snags which are mostly generic to this type of database (how would you program 'send the player through the second exit to the right from where he is now facing', for example?) but works fine. So, let's revise the method slightly and choose to store a single two-dimensional array WHERE(X,N) dimensioned DIM

WHERE(no. of rooms, 4). In this case, X still refers to the room in which the player resides and N refers to the direction he wants to travel. Thus, WHERE(X,1) would tell us which room the player reached when going in direction 1 from room X.

What's direction 1? Whatever you choose to make it! I tend to think clockwise from north, so we could define direction 1 to be north, 2:east, 3:south, and 4:west. If so, our map could be coded thus.

Second index (direction)	1	2	3	4
First index (rooms)				
1	0	0	3	0
2	0	3	0	0
3	1	4	0	2
4	0	0	0	3

It contains the same information in a different form. The entries have been flipped about a diagonal running top left to bottom right.

The problem with these systems is that they take up a lot of space: typically nearly 100 bytes for just our tiny map. Later on we shall meet up with ways of holding the same amount of information in vastly less space; the reason for doing so is not for neatness or efficiency, just to give us more space for plot!

1.4 Running programs

Let's now look at the overall running program which co-ordinates all the various activities that the program and the player will be undertaking. Devotees of structured programming will be shocked. They would argue that you should write all your programs 'top-down', like this:

```
10 PROCSETUP
20 REPEAT
30 PROCFINDOUTWHATPLAYERWANTSTODO
40 PROCTELLPLAYERWHATHAPPENS
50 UNTIL PLAYERDEAD
60 PROCDIE
```

Each of the procedures would then be written. In turn they would almost certainly involve other procedures – and so on.

This is fine for short programs, and I recommend it highly. But for anything complex a great deal of thought is involved before you can write PROCSETUP. For example, look at the room database system in the previous section. Suppose you decided on one of the versions I mentioned, and then wrote PROCSETUP. A couple of days later, while working through a procedure called PROCDESCRIBE-PLAYERSROOM, you might decide that you want some rooms (and some rooms only) without any natural light. Alas, your room database doesn't include such an option – it merely tells the computer about entrances and exits. Cursing, you decide to add an array: LIGHT(X), which is TRUE if room X is lit and FALSE otherwise. That should solve the problem. But then, a little later still, you remember that the player must carry a lantern in order to illuminate those dark rooms. So, the describing procedure should check if the player is carrying a lantern before the computer can announce whether the room is, or is not, pitch dark. Then again, suppose the player had the lantern when he entered the room but dropped it. In that case, he would not be carrying a lantern but the room would neverthless be illuminated.

Obviously this could get very complicated very rapidly, and unnecessarily so. The fact is that Adventure games need a bit of 'bottom-up' programming too, a degree of thinking about the nuts and bolts of the program before putting pen to paper.

Having said this, your final program should be very well structured – it's only a question of how the structure is produced. It is vital that you know your way around your program, and that implies a structured approach. If, when you are debugging the program, you are presented with: 'Array in line 2040', you need to know not only what that line was supposed to do (and have a copy of what the line purports to say), but also where the variable for the array was set last, what its value was supposed to be, and so on.

An example of the thought required is in your choice of variables. BBC BASIC lets you use variables with as long and as descriptive a set of names as you like. For short programs, this minimises both the writing and debugging time – as does the inclusion of plenty of REMs within the program. But these all take up space;

the variable PLAYERSROOM needs 11 characters or bytes just to mention it in the program; whereas R takes up but one character. Use the longer variable 50 times in your program and you have used up 550 characters just mentioning the player's room! On the other hand, R isn't the most descriptive of labels and hardly facilitates reference. The solution is obvious – think about the variables before you write the program, and make a list of what they are going to represent.

This becomes really important when considering 'work variables'. These are variables to enable the computer to make a quick calculation. For example, you might need to count how many rooms have been visited by the player (the array VISITED(R) would be TRUE if room R had been visited and FALSE otherwise). So you could well write something like:

```
500 I%=0
510 FOR J% = 1 TO NUMBEROFROOMS
520 IF VISITED(J%) I% = I% + 1
530 NEXT
```

which would leave I% holding the number of visited rooms (and J% with a value you don't really care about).

I used I% and J% in that example for two reasons: (a) I always use them for work variables, and (b) they are two of the resident integer variables; they don't take up any space because they exist already. However, do be careful not to use them unwisely. Programs tend to include lots of procedures which call each other (PROCA includes a line which calls PROCB, which itself contains a line which calls PROCC, and so on). If, then, we use I% and J% both as work variables in the main program and in PROCA, and include something of the order, I%=3: PROCA: IF I%=N% THEN . . ., PROCA will almost certainly have redefined I%. Think about so structuring your use of work variables that they don't get called upon to play several roles in several procedures simultaneously.

One remedy is to use LOCAL variables for the procedures. Thus DEF PROCA: LOCAL I%, J% will particularise I%, J% to PROCA. Predictably, we pay for this in SPACE. If the computer is to keep track of

the values I% & J% in the main program and their values in PROCA, it will have to store both sets of values somewhere.

My own solution is to plot out in advance which resident work variables will be used for which 'level' of procedures or functions. By 'level' I mean that a procedure like 'move player' refers to certain functions like 'room of player', for example, whereas the procedure 'room of player' makes no such reference to the 'move player' procedure. So, I use I%, J%, K%, and L% as four work variables in my main program; A%, B%, C% and D% as variables in the 'big' procedures (like moving, etc.); and E%, etc. in the procedures called by the 'big' ones, and so on down to such elementary functions as 'where in the machine I have stored the information about room 15'. This all takes work – but the payoff is the lack of work trying to figure out how H% came to be 157 when your program clearly set it in the range 1 to 4. . .

We have merely touched on the problems involved in creating the running program. The best way to come to terms with the process is to try writing a game, which is what we shall do in the next part.

2

CREATING A 'HACK-AND-SLASH' GAME: 'CAVES'

2.1 The plot

In this part of the book we will investigate a simple adventuring game in order to demonstrate how to develop a database structure, handle commands, and so on.

As a model we will look at the bones of a more complex but truly excellent Adventure called Sorceror's Cave (the publishers are now Gibsons, and its writer is Terence Donnelly).

The game presents a team of people (initially just one man) involved in an exploration of a vast cave system, which will change from game to game. You begin in an entrance cave just underground, and may explore in any one of six directions: north, east, south, west, up and down (though not all areas have exits in each direction). Exploration reveals a three-dimensional grid of caves and passages, which should be mapped in order to avoid getting lost.

In the caves lie treasures of various values, and denizens. The latter are both ordinary people, like yourself, and mythical creations such as giants, dragons, and so on. Often the denizens are guarding treasure or blocking a route you wish to take. You have

the choice, as you would in real life, of deciding to leave well alone (a good choice where dragons are concerned!), fighting them (and gaining the advantage of surprise) or approaching them to see if they're prepared to explore with you (which will strengthen your party for later encounters). In the latter case, the denizens' leader will determine whether he likes the look of you, doesn't care one way or the other, or wants to fight you (whereupon the denizens get the advantage of surprise). Some denizens are more friendly than others. . .

Fighting is carried out by the program, not by the player, and involves a comparison (or weighing up) of fighting strengths. Each character in the game has a fighting strength; weak characters like hobbits have little strength, whereas fearsomely strong characters like dragons resemble a mobile army. The fighting strengths of all denizens present are added together and pitted against the fighting strengths of up to three of your party (since caves are awkward places, not too many people can muscle in on the fight!). There is a bonus of one for whichever side has the surprise advantage. To each of these numbers is added a random dice throw (i.e. a number between one and six). The team with the higher number kills one of the other side. In the event of a draw, you are deemed to be still fighting. This gives you the chance of running for an exit, or continuing to fight.

Only if you kill all the denizens, or if they join your party, can you pick up the treasure in their area. There are more treasures (and more denizens!) in the lower levels of the caves; thus the surface levels are safer if less rewarding.

Each area is either a cave with something inside it, or an empty passage. The type of area, its exits, and contents if any, are determined randomly the first time the player attempts to enter. Thus the player may be in a passage with a north exit but be unable to use it because the area to the north doesn't have a southern entrance.

Finding and using an 'up' staircase on a level of the caves just underground will take you out of the caves, and finish the game. Things are seldom that simple because caves and passages get blocked easily, and there are two further random events which can ruin

your plans. One is an earthquake, which will destroy the area you were just in, and render it impassable. The other, a trap, is a precipitous drop one level deeper into the cave system. In either of these cases, there is no possibility of retreat should you encounter any denizens.

That's roughly the plot. In the original, access to the next cave or passage was determined by revealing a card with a representation of a cave and some exits drawn on it, followed by a number of cards representing its contents. In our implementation, this will be replaced by random selection within the computer. However, there will have to be a fair amount of book-keeping so that the player may backtrack and rediscover caves (and where appropriate, contents) that have already been mapped.

2.2 Planning the game – the game logic

Having decided on the plot of the game, the next thing is to organise its logical flow, turn by turn. I strongly recommend doing this in English, or a quasi-English most programmers know as 'pseudo-code'. What we will do is write the program in readable English, but in terms that are converted, with relative ease, to BBC BASIC.

Since BBC BASIC is highly structured, and we aren't going to be short of room for this program, we can set up a very structured program; please bear this in mind as we proceed.

First of all, when writing pseudo-code,

GET THE LOGICAL STRUCTURE FIRST

then figure out later how to program it. Only if the programming is clearly beyond your abilities should you redesign the structure!

We'll assume that outside the main program loop there will be some initialisation, dimensioning, screen mode choosing, etc., and concentrate now on the recurring logic. As he takes a turn, the player may be in a normal, 'what shall I do now?' situation, or he may be 'still fighting' from his previous turn. Obviously the latter will take precedence over the former. So we begin our pseudo-code with:

IF PLAYER IS FIGHTING, MAKE THE AREA UNSAFE, SET FIGHT BONUS TO ZERO, AND ASK PLAYER IF HE WISHES TO CONTINUE TO FIGHT OR TO LEAVE. IF CONTINUING, FIGHT AND GO TO END OF LOOP.

Programmers will notice this 'fighting' test is going to be used time and again in the program – it is of course part of the database structure. Since we will have to write it many times, let's decide now to handle it, and questions like it, by a simple FLAG. This is a logical variable (e.g. FIGHTING) which when true will indicate that fighting is proceeding and when false indicates the opposite. This shortens the above pseudo-code to:

IF FIGHTING, SAY SO, SET UNSAFE, BONUS=0. ASK PLAYER 'CONTINUE OR LEAVE?' IF CONTINUE, FIGHT; GO TO END OF LOOP.

Notice how 'Basic-like' this looks. It makes it more likely that when the programming begins, we will get it right (or nearly right) the first time rather than the tenth time!

Now, what's it all about? Well, if the player is fighting, we shall remind him, and jot down that the area is 'unsafe'. Later on, if an area is unsafe, it will stop the player grabbing treasure, checking the status of his party, and all the other things which wouldn't make sense in the presence of – not necessarily hostile – strangers. We then set the fight bonus to zero, as in a continuing fight nobody has a surprise advantage. Then we check whether the player is continuing the fight, or leaving. If he fights, let him do a round, and then end the loop. (If he leaves, he'll just continue the main program loop.)

Now we can proceed with the normal course of events. First, tell the player where he is and find out what he wants to do next:

DESCRIBE ROOM. ASK PLAYER FOR A DIRECTION TO MOVE (IF SAFE, ALSO ASK IF HE WISHES TO KNOW HIS PARTY'S STATUS).

After doing this, we have to examine whether he can go in that direction. Many things can intervene, and this is where a little logical thinking comes in. Put

yourself in the player's position for a moment. Let's say you've just said 'north'. What could stop you? First, there might be no north exit in your area. (I can hear you objecting that we've just described the room, so the player knows there's an exit north. Well, he's allowed to make mistakes, and this might be one of them.) So let's write (incorrectly at first, as we'll see):

IS THE EXIT BLOCKED? IF SO, SAY SO, AND GO TO END OF MAIN LOOP.

Stop a moment and be the player again. If he wasn't fighting, that's fine. But if he was fighting, his inability to leave would plunge him back into the melee again. So we have to modify this outcome to:

IS THE EXIT BLOCKED? IF SO, SAY SO. IF FIGHTING, FIGHT AND GO TO END LOOP, ELSE GO TO END LOOP.

A word of warning here: in BBC BASIC the first 'ELSE' encountered on a line pairs with the first 'IF' before it on the line that failed. If you think like me, though, that 'ELSE' refers to 'IF NOT FIGHTING'. The whole clause, therefore, refers to what happens if the exit is blocked. Note also that we always tell the player something (user-friendliness!).

Now we know the player can leave his room, what else may happen? First of all, he may be trying to exit the maze with his treasure, which he can do if he's on the top level and is moving up. On the other hand, he may not have noticed which level he's on. Hence we should give him the choice:

WILL THIS TAKE PLAYER OUT OF MAZE? IF SO, SAY SO AND OFFER THE CHOICE OF NOT LEAVING. IF LEAVES, SCORE, END GAME, AND OFFER A NEW ONE; ELSE IF FIGHTING, FIGHT; GO TO END LOOP (ELSE JUST GO TO END LOOP ANYWAY).

Notice that awkward test for fighting recurring; life's like that. Next we must check whether there's already a room in existence for the player to go to, or whether we have to design a new room and its contents. Since any database must be of a limited size, we shall choose to allow only 90 rooms in total. So it may not be possible to design a new room:

IS THERE A ROOM ALREADY THERE? IF NOT, TRY TO MAKE AND FILL A NEW ROOM. IF THERE'S A PROBLEM, REPORT THE EXIT TO BE BLOCKED AND CHECK FOR FIGHTING AS USUAL BEFORE GOING TO END LOOP.

So now there's a room for the player to go to – but there may be no entrance. Let's check:

IS THERE AN ENTRANCE FOR THE PLAYER IN THE NEXT ROOM? IF NOT, REPORT BLOCKAGE, FIGHT IF NECESSARY, AND GO TO END OF LOOP.

Believe it or not, after all this the player is ready to be moved! If you've programmed up similar things before, none of the foregoing will have been unusual. If you're new to this, you may still be puzzled as to how you would generate an equivalent set of questions in your own games. It's a knack, really. Try to think negatively rather than positively. Put yourself in the player's position. If there were no problems, he'd just move. So assume there are problems. Before you start writing in pseudo-code, don't worry about getting them in logical order, unless you have that kind of mind! Just jot them down on a piece of scrap paper as they occur to you, until you're convinced you have a complete list. Then put them into sequence. There's no point in checking, say, the entrance to the next room until you've ascertained whether the player can leave his current one, and so on.

OK, back to the plot, but still thinking negatively. The player can move, but something may well happen to him on the way: those random events, namely earthquakes and traps. Both of these would stop him retreating from his new room. So we first set a flag which will let him retreat, and then if an event occurs to stop him we can cancel the retreat:

SET RETREAT OK.
CHECK FOR EARTHQUAKES.
IF EARTHQUAKE, SAY SO; SET CURRENT ROOM TO RUBBLE; SET NO RETREAT, AND SKIP OVER THE 'TRAP' SEQUENCE.
CHECK FOR TRAPS.
IF TRAP, TRY TO MAKE NEW ROOM BELOW CURRENT ONE. IF CAN'T, IGNORE TRAP; ELSE SAY SO, REDEFINE NEW ROOM TO BE ONE BELOW AND SET NO RETREAT.

Now we simply shift the player:

MOVE PLAYER TO NEW ROOM.

What problems can afflict him once he's in the new room? Perhaps none: the room may be empty:

IF ROOM EMPTY, SET NOT FIGHTING, SET SAFE, SET RETREAT OK,
GO TO END LOOP.

That gets the easy case out of the way! The loop will circle round to the beginning and do a description of the room, which is what is needed in this position. But suppose there is something in the room. We should first tell the player what it is, then see if he likes what he sees:

DESCRIBE ROOM.
IF RETREAT POSSIBLE AND THERE ARE DENIZENS, ASK IF WISHES TO RETREAT. IF SO, RETURN TO PREVIOUS ROOM (AND FIGHTING IF NECESSARY) AND END LOOP.

Note that check for fighting again – we don't want this to be too easy! We obviously don't ask about retreating if there aren't any denizens, because in that case there's only treasure, which the player will pick up automatically in a moment. First we must check the SAFE flag:

IF ANY DENIZENS UNSET SAFE ELSE SET SAFE

(because this might be left over as unsafe from the previous turn). Now the player may be able to pick up the treasure:

IF SAFE, EMPTY ROOM OF TREASURE AND GO TO END LOOP.

So if we get here in the logic, there must have been some denizens (i.e. it was UNSAFE). Now the player gets the choice of approaching them or fighting them:

ASK IF PLAYER WISHES TO APPROACH OR TO FIGHT.

Now the logic forks, depending on what he elects to do. Let's look at approaching first. Three outcomes are

possible: the denizens may choose to join the party, ignore the party, or attack it with the advantage of surprise. All we have to do here is to note the outcomes in each case, and program up the mechanics later:

IF APPROACH:
IF DENIZENS JOIN, SAY SO, ADD THEM TO PARTY AND EMPTY ROOM (I.E. GET THE TREASURE); GO TO END LOOP.
IF DENIZENS IGNORE PARTY, SAY SO AND GO TO END LOOP.
IF DENIZENS ATTACK PARTY, SET BONUS FOR DENIZENS, FIGHT, AND GO TO END LOOP.

This leaves only the fight option, which scores the bonus for the party:

IF FIGHT:
SET BONUS FOR PARTY; FIGHT; END LOOP.

And that is the end of the program logic. Before we discuss how we might store and access all the information the program will need, here's a repeat of the logic in continuous sequence. You may find it useful to refer to when we do the programming.

IF FIGHTING, SAY SO, SET UNSAFE, BONUS=0. ASK PLAYER "CONTINUE OR LEAVE?" IF CONTINUE, FIGHT; GO TO END OF LOOP.

DESCRIBE ROOM.

ASK PLAYER FOR A DIRECTION TO MOVE (IF SAFE, ALSO ASK IF HE WISHES TO KNOW HIS PARTY'S STATUS).

IS THE EXIT BLOCKED?
IF SO, SAY SO. IF FIGHTING, FIGHT AND GO TO END LOOP, ELSE GO TO END LOOP.

WILL THIS TAKE PLAYER OUT OF MAZE?
IF SO, SAY SO AND OFFER THE CHOICE OF NOT LEAVING. IF LEAVES, SCORE, END GAME, AND OFFER A NEW ONE; ELSE IF FIGHTING, FIGHT; GO TO END LOOP (ELSE JUST GO TO END LOOP ANYWAY).

IS THERE A ROOM ALREADY THERE?

IF NOT, TRY TO MAKE AND FILL A NEW ROOM. IF THERE'S A PROBLEM, REPORT THE EXIT TO BE BLOCKED AND CHECK FOR FIGHTING AS USUAL BEFORE GOING TO END LOOP.

IS THERE AN ENTRANCE FOR THE PLAYER IN THE NEXT ROOM?
IF NOT, REPORT BLOCKAGE, FIGHT IF NECESSARY, AND GO TO END OF LOOP.

SET RETREAT OK.

CHECK FOR EARTHQUAKES.
IF EARTHQUAKE, SAY SO; SET CURRENT ROOM TO RUBBLE; SET NO RETREAT, AND SKIP OVER THE 'TRAP' SEQUENCE.

CHECK FOR TRAPS.
IF TRAP, TRY TO MAKE NEW ROOM BELOW CURRENT ONE. IF CAN'T, IGNORE TRAP; ELSE SAY SO, REDEFINE NEW ROOM TO BE ONE BELOW AND SET NO RETREAT.

MOVE PLAYER TO NEW ROOM.

IF ROOM EMPTY, SET NOT FIGHTING, SET SAFE, SET RETREAT OK, GO TO END LOOP.

DESCRIBE ROOM.

IF RETREAT POSSIBLE AND THERE ARE DENIZENS, ASK IF WISHES TO RETREAT. IF SO, RETURN TO PREVIOUS ROOM (AND FIGHTING IF NECESSARY) AND END LOOP.

IF ANY DENIZENS UNSET SAFE ELSE SET SAFE

IF SAFE, EMPTY ROOM OF TREASURE AND GO TO END LOOP.

ASK IF PLAYER WISHES TO APPROACH OR TO FIGHT.

IF APPROACH:
IF DENIZENS JOIN, SAY SO, ADD THEM TO PARTY AND EMPTY ROOM (I.E. GET THE TREASURE); GO TO END LOOP.
IF DENIZENS IGNORE PARTY, SAY SO AND GO TO END LOOP.

IF DENIZENS ATTACK PARTY, SET BONUS FOR DENIZENS, FIGHT, AND GO TO END LOOP.

IF FIGHT:
SET BONUS FOR PARTY; FIGHT; END LOOP.

2.3 Planning the game – the database structure

Having organised the playing logic, we must now decide how the various details (the cave layout, the contents, who is in the player's and so on) can be stored. In other words, decide on the database structure.

There are many ways we might choose to store the information. Not all are as efficient as others. Let's start with the cave layout,and leave the cave contents until later. The cave system is built up on a three-dimensional grid, so that a map of the system would need three dimensions: north-south, east-west, and up-down. Unlike the map we looked at in Part 1, it's determined randomly every game (we can't worry about how just yet as we don't even know how to store it!). So for each area in the system, we need to establish two things: (a) which directions are exits, and (b) where in the 3-D grid the area lies.

That may not be obvious, but think about it for a moment. We can't store the destination rooms for the six possible exits to any given room, because as yet we cannot identify the room or even tell whether any exist exist exist. So that precludes the system in Part 1. (I suppose one could use an entry of '0' to mean 'exit blocked', a number from 1 to 90 to indicate which room lay beyond the exit, and a 99 to mark 'no room yet in existence there', but it's a little clumsy and, as we shall see, space-consuming.) Hence the method suggested. If the player tries to go north, say, we can look up in (a) to see if that's possible, then check where he is in (b), and decide whether there's a room to the north by using (b) again. After making an exit, if necessary, we can check to see if that room has an exit to the south (to let the player in), again by using (a).

The first essential is to identify the rooms. Let's number them as in Part 1. The player won't know they are numbered, of course, but the program will. Also, let whichever room the player is in be accorded the integer variable "r%". (For simplicity all the variables used, except local work variables, will be lower case

integers; all logical variables and arrays will be upper case for this part.) So the directions with exits from room r% can now be stored in one or more arrays.

And therein lies the first snag. Suppose we create six arrays: NORTH(90), EAST(90), etc. (because there are 90 rooms). The entry in NORTH(10) will be zero if there is no exit northwards from room 10, and one if there is such an exit. Well, try it. Turn on your computer, and if your BBC BASIC is version 1 or 2, type the (non-Acorn supported)

```
PRINT !2 AND &FFFF
```

(This little bit of gobbledegook tells you where the first available piece of memory is, after your program variables. When this value hits about HIMEM, you get that awful 'No room' message.) Jot down the answer, then type:

```
10 DIM NORTH(90),EAST(90),SOUTH(90),WEST(90),UP(90),DOWN(90)
```

followed by RUN. This simply reserves space in memory for these six arrays. If you now type

```
PRINT !2 AND &FFFF
```

again, you should find a number about 2790 higher than it was before. In other words, simply dimensioning those six arrays used up nearly 3000 (i.e. 3K) of precious memory. Declaring them as integer arrays helps a little, but not a great deal. Thus the EASY way to store the information is a WASTEFUL way. And obviously so. Each of the 5 bytes which one entry of NORTH occupies will hold either a one or a zero; hardly an optimal use of space!

To save space drastically – at the cost of working a little harder on the programming – we can store the entire exit information in one array, called R%(90). It's important that it be an integer array, as you'll see in a moment; in fact, very little in adventure games needs real numbers. Suppose that the exits from room 13 are north, south and down. Then we can make the value of R%(13) be 100101, where we use the code:

```
S W U D E N
1 0 0 1 0 1
```

(an integer variable can be any value up to about 2 thousand million). Each digit of R%(13) then holds a flag to indicate whether there is an exit in the appropriate direction. I'll come to my reasons for choosing that particular order of directions later.

We haven't yet exhausted the size of R%, since the code scheme only uses 6 digits. It seems a pity to waste a digit, let's determine that the millions digit will be a 1 if the room is a cave (and hence has contents, at least when it's created) or a 0 if the room is a passage (without contents). So if room 13 is a cave, R%(13) becomes 1100101. Similarly, if R%(14) is 11010, we interpret room 14 as being a passage (because R%(14) has no millions digit), and with exits west, up, and east, to correspond with the 1's in its value. Be sure to grasp this because we'll be using similar ideas later.

The point of this apparently complicated setup was, you'll recall, to save space. The entire information about exits, together with a flag marking whether a room is a cave or a passage, is summarised in ONE integer variable, which takes up only 4 bytes, instead of the 24-30 we were using previously. (In Part 4, I'll show you how to get this down to one-and-a-half bytes!) But we shall pay for this saving by having to write a program to find out whether there's a northward exit or not, instead of just looking at the array NORTH.

By the way, you may have noticed there are several digits left over in R% which could have been used to store information. You may want to modify the program afterwards to add some frills of your own, and there are three digits left you can play with.

Now to store the location of room r%, again in a single array (slightly too large to fit into the remaining digits of R%) called WHERE%(90). If room 13 is 4 levels down from the surface, 8 areas east and 47 areas north in the three-dimensional grid, then WHERE%(13) will be 40847. We use the scheme:

```
 D  E  N
04 08 47
```

to encode the location numbers for room 13. Such numbers are known mathematically as the room's co-ordinates. Hence if WHERE%(14) is 174923, room 14 is 17 levels down, 49 areas east, and 23 areas north. We have to use 2 digits here as the player may well roam quite some distance.

Before we accept this scheme, think negatively. What if the player goes south successively until his

'north' co-ordinate becomes 2, then 1, then zero, and then . . . ? We can't place a negative number for the north co-ordinate in the middle of a normal six-digit number, so instead we must define all ways as blocked when north or east co-ordinate reaches what would be a negative number. To circumvent this altogether, we could define the entrance cave as east 45, north 45 (chosen merely to be half of 90, by the way). Then, the down co-ordinate will never be a problem since the entrance cave is at level 1, and the player can only go deeper; also, you will recall that any move to rise above level 1 will end the game, so no worry there; and, finally, the player will never get 45 rooms in one direction, I guarantee! (If you are paranoid about this, put a check into the program later.)

The only other big array should seem fairly straightforward now. We need one to hold the contents of each cave – or indeed, each area, even though a passage won't have any contents. We call this CONTENTS%(90), and let any area hold as many as four things, each of which can be a treasure or a denizen. (We'll decide how to number those in a moment.) So if CONTENTS%(13) = 12060312, then room 13 holds things number 12, 6, 3, and 12 again. We take pairs of digits just as before:

```
Thing   1  2  3  4
       12 06 03 12
```

and another example would be CONTENTS%(14)=0, which would mean there are no things in room 14. (Thing 2 can be zero, while things 1 and 3 are not zero, by the way.)

There will also need to be some smaller arrays connected with denizens. The time has come to decide which denizens we shall have in the game. My list contains 10 different types. These are: dragon, giant, wizard, orc, man, woman, dwarf, hobbit, wolf, and lion. These are numbered from 1 to 10 in order of importance when it comes to deciding which denizen chooses whether to join the player's party (i.e if there's a dragon present, he chooses). We shall also have five possible treasures: bronze, silver, gold, gems, and a treasure chest, in increasing order of value. There's nothing significant about 10 and 5 – change the numbers of denizens or treasures if you wish. In order to be able to print out what is present in an area, we

make a string array to hold the names of these 15 things, called CHAR$(15). CHAR$(1) = 'A dragon', and so on.

Another array for things is needed to hold their individual characteristics. For each denizen, we need to be able to refer to its fighting strength, and how likely it is to attack or to befriend the party. The strength will be a number from 1 to 10 (again, there's no specific reason for the limits). The likelihood of attacking or befriending is handled in a similar manner to the original game: each denizen has associated with it two numbers from 1 to 6, namely its attack threshold and its befriending threshold. When approaching the most important of the denizens in an area, a die is thrown (i.e. a random number between 1 and 6 is generated). If the number is less than or equal to the attack threshold, the denizens attack. If the number is greater than or equal to the befriending threshold, the denizens befriend. If the number lies in between the two, the denizens remain neutral. This enables us to make dragons impossible to befriend – their befriending threshold will be 7 – and almost impossible to avoid fighting – their attack threshold will be 5. Conversely, little peacable hobbits have an attack threshold of 1 and a befriend threshold of 3. We hold these numbers in a single array called CHAR%(15); the remaining 5 elements will be used for the value of the five treasures, with CHAR%(11) for the bronze, up to CHAR%(15) for the treasure chest. We pack in the fighting strength and the two thresholds into a single number just like CONTENTS%, with three sets of two digits. Hence CHAR%(1), for the dragon, is 70510, which means:

```
Be   At   St
07   05   10
```

where Be, At, and St stand, of course, for Befriending, Attacking, and Strength respectively.

Two more arrays complete the list. The first is TEAM%(30), whose entries are the numbers of the characters in the party. TEAM%(1), then, is 5, because the first member of the party is a man, numbered 5 on the list of 10. TEAM%(2) is undetermined so far, but will be set when a denizen joins the party. The last array is for printing convenience only: DIRN$(5). Each of the 6 entries (we're using DIRN$(0) here also, for programming convenience) contains a string

corresponding to one of the six directions, in the reverse order to the way R% stores directions. Thus DIRN$(0) = 'North', DIRN$(1) = 'East', and so on up to DIRN$(5) = 'South'.

These, plus a few variables, are all we need to program the game.

2.4 Programming – the main program

Thanks to our pseudo-coding the game logic, the main program loop now almost writes itself. We need merely to set a screen mode, do some dimensioning, call a setting-up procedure, and then loop around the logical structure we've created. This gives us (in chunks, to allow for explanations):

```
  10 DIM R%(90),WHERE%(90),CONTENTS%(90),TEAM%(30),CHAR%(15),CHAR$(15),
DIRN$(5)
  20 MODE 7:PROCSETUP
  30 REPEAT SNAG=FALSE
  40 IF FIGHTING SAFE=FALSE:bonus%=0:PRINT"Do you wish to Continue
fighting (C)"'"or Leave (L) ?":REPEAT A$=GET$: UNTIL A$="C" OR A$="L":IF
A$="C" PROCFIGHT:UNTIL FALSE
  50 CLS:PROCDESCRIBE
  60 PROCGETDIRECTION
  70 PROCEXITBLOCK
  80 IF SNAG AND FIGHTING PROCFIGHT:UNTIL FALSE FLSE IF SNAG UNTIL FALSE
  90 PROCLEAVEMAZE:IF SNAG AND FIGHTING PROCFIGHT:UNTIL FALSE ELSE IF
SNAG UNTIL FALSE
 100 PROCROOMTHERE(1)
 110 IF SNAG AND FIGHTING PROCFIGHT:UNTIL FALSE ELSE IF SNAG UNTIL FALSE
```

Here are the explanations.

10: Dimension all needed variables.

20: Set mode 7 (i.e. 6 on the Electron) for needed space; call PROCSETUP (to be written) to initialise variables.

30: Enter main loop, and clear a flag SNAG, which is set to true by most of the procedures if, not surprisingly, a snag occurs for the player. This trick of passing back a flag from a procedure avoids lots of awkward programming.

40: A long line, but containing a single piece of logic. If fighting, find out player's decision. (Note the use of GET$ – the only input method used in the game – and the way it is checked. In particular, we're only accepting upper case input. How could you allow for lower case as well?) If he wishes to leave, the program will simply continue. If he wishes to fight, PROCFIGHT will be called (not written yet!) followed by the very useful UNTIL FALSE. The main loop's

REPEAT can have many UNTILs to end it with; here is the first.

50: Clear screen and describe room and contents.

60: Ask for direction to move (and/or party's status if SAFE).

70-80: If exit blocked (marked by SNAG) fight or not depending on FIGHTING. Care is needed with the IF–ELSE ordering here.

90: Leaving maze check. SNAG will be set in PROCLEAVEMAZE if player decides not to leave. You could if you preferred have used functions instead of procedures, and written SNAG=FNLEAVEMAZE. There's no 'right' and 'wrong' way to do all this!

100-110: Is a room there for player to go to? Note the passing of a marker 1 here, as we'll use a similar procedure if the player falls down a trap, but don't want to set SNAG in that case. The procedure will here make a room if necessary. SNAG will get set only if a room is impossible to make, and will generate fighting as per usual.

```
  120 PROCENTERROOM
  130 IF SNAG AND FIGHTING PROCFIGHT:UNTIL FALSE ELSE IF SNAG UNTIL FALSE
  140 RETREAT=TRUE
  150 PROCEARTHQUAKE
  160 IF NOT SNAG PROCTRAP
  170 SNAG=FALSE
  180 rold%=r%:r%=r1%
  190 IF CONTENTS%(r%)=0 FIGHTING=FALSE:SAFE=TRUE:RETREAT=TRUE:UNTIL FALSE
  200 PROCDESCRIBE
  210 IF NOT RETREAT OR NOT FNANYDENIZENS ELSE PRINT"Do you wish to
retreat (Y/N) ?":REPEAT A$=GET$:UNTIL A$="Y" OR A$="N":IF A$="Y" AND
FIGHTING r%=rold%:PROCFIGHT:UNTIL FALSE ELSE IF A$="Y" r%=rold%:UNTIL
FALSE
  220 IF FNANYDENIZENS SAFE=FALSE ELSE SAFE=TRUE
  230 IF SAFE PROCEMPTYROOM:FIGHTING=FALSE:UNTIL FALSE
  240 PRINT'"Do you wish to Approach (A)"'"the denizen(s) or Fight
(F)?":REPEAT A$=GET$: UNTILA$="A" OR A$="F"
  250 IF A$="F" ELSE PROCAPPROACH:IF NEUTRAL UNTIL FALSE ELSE IF SNAG
PROCFIGHT:UNTIL FALSE ELSE UNTIL FALSE
  260 bonus%=1
  270 PROCFIGHT
  280 UNTIL FALSE
```

Explanations:

120-130: Is there an entrance? If not, fight etc. as needed.

140: The player is moving whether he likes it or not! Let him retreat unless something awkward happens.

150-160: Test for earthquakes. (If there was one, the procedure will reduce the current room to rubble; in that case don't burden the player with a trap as well.) Test for traps. The procedure will alter the destination

room to the one below in this case. (r1% by this time holds the destination room number.)

170-180: Clear SNAG to avoid problems, and move the player. This simply makes the value of r% be r1%, the destination! At the same time, to allow for possible retreats, remember where player came from in rold%.

190: If the room is empty, make all safe and end the loop.

200: Not empty, so tell player what there is. Note re-use of an earlier procedure.

210: If there are any denizens and the player can retreat, find out if he wishes to. If so, return him (r% = rold%) and check for fighting again. This line has more than its share of IFs. In particular, we only want to do a retreat check IF two conditions are satisfied, yet there will be other IF-ELSE checks in the line. We could avoid ELSE conflict by using a GOTO, but structured programming frowns on this (tough luck -we'll be using them later!). So instead we use the fact that BBC BASIC does not require a THEN clause at all (merely passing on to the next line if the condition is satisfied). This kills the first ELSE early on and allows more IFs later in the line.

220: If the room is inhabited, make it unsafe.

230: If the room is safe, collect the treasure – which will leave the room empty.

240: The room was inhabited. Find out player's decision.

250: He's approaching (note the IF-ELSE trick again). Take action if neutral or fighting (approach procedure will handle joining).

260-280: He's fighting. Set bonus, fight, and end loop.

Working out the logic of the game in section 2.2 not only takes care of a great deal of the programming work, but also minimises the incidence of errors too. Of course there'll be bugs in individual procedures – there were when I wrote these! – but they're easily traceable, as I'll show you when I talk about debugging later.

2.5 Programming – the procedures (1)

Now comes the pleasant part of the programming – writing the procedures and making the program come 'alive'. Unless there are pressing needs otherwise, it's usually simpler to write the procedures in the order they appear in the program, and I'll adopt that here. You'll already have found in your own programming

that it pays to use fairly separated line numbers for procedures (to allow for debugging later) so we'll start at line 500, with PROCSETUP.

```
  500 DEFPROCSETUP
  510 nroom%=1:r%=1:R%(1)=1111011:WHERE%(1)=14545
  520 nteam%=1: TEAM%(1)=5:score%=0
  530 FOR I%=1 TO 15:READ*CHAR%(I%),CHAR$(I%):NEXT
  540 DATA 70510,"A dragon",60406,"A giant",50304,"A wizard",50405,"An
orc",40203,"A man"
  550 DATA 40202,"A woman",30101,"A dwarf",30101,"A hobbit",50304,"A
wolf",50305,"A lion"
  560 DATA 2,"Some bronze",5,"Some silver",10,"Gold bars",20,"Gems",30,"A
treasure chest"
  570 DIR$="":FORI%=0 TO 5:READ DIRN$(I%):DIR$=DIR$+
LEFT$(DIRN$(I%),1):NEXT
  580 DATA "North","East","Down","Up","West","South"
  590 FIGHTING=FALSE: SAFE=TRUE: RETREAT=TRUE
  600 ENDPROC
```

Explanations:

510: Only one room so far; player is in room 1; room 1 is a cave (without contents) with exits south, west, up, east and north; it lies on level 1, 45 rooms east and north.

520: One member in the party, who's a man (we get the numbering system next line); no score so far.

530: Read and store characteristics and names of all things which can be met in the cave system, using DATA lines 540 to 560. To remind you of the storage system, the dragon has a befriending threshold of 7, an attacking threshold of 5, and a fighting strength of 10. The bronze treasure is simply worth 2.

570: Using the DATA in 580, read in the six directions. Store their initial letters – in capitals – in DIR$, which finally equals "NEDUWS". This order is exactly the reverse of the way we code directions in the database, for a reason you'll appreciate soon. DIR$ will be used as an efficient check to see if players type in the correct response to a directional request.

590: Set various flags to their initial values. All the other dimensioned variables will be set to zero anyway by BASIC.

The next procedure to be met is PROCFIGHT, which occurs many times in the program. I'm going to skip over it for now, as the only time it occurs without a preceding IF check is towards the end of the main loop. The beauty of procedure-writing is that you can check as you go, so it's easier to write only the ones we need at any time. So we move on to PROCDESCRIBE.


```
  700 DEFPROCDESCRIBE: LOCAL I%
  710 I%=R%(r%) DIV 1000000:IF I% A$="cave" ELSE A$="passage"
  720 IF r%=1 PRINT''"You are in the entrance cave with exits:" ELSE
PRINT''"You are in a ";A$;" with exits:"
  730 FOR I%=0 TO 5
  740 IF FNEXIT(I%,r%) PRINT DIRN$(I%)
  750 NEXT
  760 IF CONTENTS%(r%)=0 ENDPROC
  770 PRINT'"In the cave is:"
  780 FOR I%=0 TO 6 STEP 2
  790 J%=FNWHO(r%,I%)
  800 IF J%>0 PRINT CHAR$(J%)
  810 NEXT
  820 ENDPROC
```

Explanations:

710: Is the area a cave or a passage? The information is in R%(r%), and depends whether the millions digit is a 1 or a 0. A quick way to find out is to integer divide by a million (much quicker than normal dividing). I% is a standard work variable. Then we set A$ (a working string) to the type of area. Notice that we can just say IF I%, since a non-zero I% yields TRUE anyway.

720: Describe the area. A special test to see if we are in the entrance cave (the first room).

730-750: Scan the six directions (0 = North, up to 5 = West) and see if there is an exit. We use a logical function FNEXIT(I%,r%) for this. FNEXIT returns a value of TRUE if room r% has an exit in direction I%, and FALSE otherwise. More on this in a moment, when we write it.

760: If area empty, end description.

770-810: Otherwise, check all four possible things in the cave, and write out their names. Here we use another function, FNWHO(r%,I%). This gives the (I%/2)th thing in room r%. (We use counting by twos merely because WHERE% holds thing labels in two digits.) If any thing exists (i.e. J% > 0), print it out.

This procedure referenced two functions. The first is FNEXIT, which will be used by various procedures. FNEXIT(dir%,r%) will be TRUE if there is an exit from room r% in direction dir% (0 to 5) and FALSE otherwise:

```
900 DEFFNEXIT(I%,r%):LOCAL J%
910 J%=(R%(r%) DIV 10^I%) MOD 10
920 IF J%=1 THEN =TRUE ELSE =FALSE
```

Explanations:

910: We must examine one of the digits of R%(r%) to see if it is 0 or 1. If the relevant exit is, say, Down, this is digit 2 in our scheme, or the 10's digit in ordinary

arithmetic. If the exit were West, I% would be 4, so we would have to examine the 10,000's digit. We do so by stripping off all the other digits and leaving just one to examine. First we integer divide by a suitable power of 10, to 'lose' all the digits to the right of the one we want. For Down, we want to lose the units and 10s digits, so we divide by 100:

```
     R%(r%)= (say) 1 1 0 1 1 0 1
we want this digit        ▲
   R%(r%) DIV 100= 1 1 0 1 1
```

and in general we divide by 10 to the power of I%. For non-mathematicians, 10 to the power 0 is 1, so we can even get the North digit this way. Now we have to remove the digits to the left of the one we want, so we use the BASIC MOD function, which throws away multiples of any number. In this case we can lose all multiples of 10, so we MOD with 10. This gives, continuing the Down example,

```
          R%(r%) DIV 100=  1 1 0 1 1
       we want this digit          ▲
(R%(r%) DIV 100) MOD 10=           1
```

So J% here = 1.

```
920: Return TRUE or FALSE as required.
```

This may have sounded rather complicated, but remember we are trading programming effort for space. If ever in doubt, test the procedure on any data, as you write it. There are other ways of picking up specific digits without arithmetic – converting R%(r%) to a character string via STR$, then using MID$ to locate the specific digit. I didn't use these because the later methods we'll use will store rather more information than just one number in a single digit, in which case STR$ isn't much use. Now for FNWHO:

```
950 DEFFNWHO(r%,I%)=(CONTENTS%(r%) DIV10^I%) MOD 100
```

The logic here is rather like FNEXIT. We have to look at CONTENTS%(r%) and pick out a pair of digits beginning at digit I% from the right. (The rightmost digit, the units digit, is I%=0.) First we remove all digits further right, if there are any, by dividing by 10 to the power I%. Then we kill all digits beyond (what are now) the units and 10's columns by MODding with 100.

Now comes PROCGETDIRECTION. This is fairly straightforward:

```
1000 DEFPROCGETDIRECTION
1010 PRINT'"What would you like to do:"
1020 FOR I%=0TO5
1030 PRINT "Move ";DIRN$(I%);" (";MID$(DIR$,I%+1,1);")"
1040 NEXT
1050 IF SAFE PRINT "Condition of party (C)"
1060 PRINT "?"
1070 REPEAT
1080 A$=GET$
1090 IF SAFE AND A$="C" PROCSTATUS:UNTIL FALSE
1100 J%=INSTR(DIR$,A$):UNTIL J%>0
1110 dir%=J%-1:ENDPROC
```

Explanations:

1010: Ask player.

1020-1040: List the six directions (be user-friendly!). Pull the single letters out of DIR$.

1050: Allow status check if safe

1060: Cue player to type something!

1070: Begin a REPEAT loop until we get a useful key pressed.

1080: Read a key, using the A$ work variable.

1090: If status check is permissible, call that procedure and continue the loop with UNTIL FALSE (as we haven't got a direction yet).

1100: The use of DIR$ now becomes clear. We look to see if A$ is one of the letters in DIR$. If not, try again.

1110: Set the direction (dir%) to be one less than the position of A$ in DIR$, as INSTR counts from 1 but we are counting from 0, and return.

We referred to PROCSTATUS in the above, and this is easy:

```
1200 DEFPROCSTATUS:LOCAL I%
1210 PRINT '"Your team consists of:"
1220 FOR I%=1 TO nteam%
1230 PRINT CHAR$(TEAM%(I%))
1240 NEXT
1250 PRINT '"Your score is now ";STR$(score%)
1260 PRINT"Where will you move now?"
1270 ENDPROC
```

Explanations:

1220-1240: Print out the name of each person in the team. The I%th person has a number TEAM%(I%) – call this number J, say – and the label of the Jth type is CHAR$(J).

1250: Print the score. Note the use of STR$ to make the format pretty.

1260: Remind player he's got to move, and return to

PROCGETDIRECTION.

The next item involves checking for a blocked exit. We can use FNEXIT again here:

```
1400 DEFPROCEXITBLOCK
1410 SNAG=NOT FNEXIT(dir%,r%)
1420 IF SNAG PRINT'"You can't go that way!":PROCDELAY
1430 ENDPROC
```

Explanations:

1410: Set SNAG TRUE or FALSE opposite to FNEXIT, by use of NOT.

1420: If no exit, say so, and give player time to read it.

The delay loop is a common feature in computer displays. This one is nothing fancy:

```
1500 DEFPROCDELAY: LOCAL N%
1510 N%=TIME:REPEAT UNTIL TIME > N%+100
1520 ENDPROC
```

Now we must check if we are about to leave the maze:

```
1600 DEFPROCLEAVEMAZE
1610 IF dir%<>3 OR WHERE%(r%)>=20000 ENDPROC
1620 PRINT'"That will take you out of the maze."'"Do you really want to
eave? (Y/N)"
1630 REPEAT
1640 A$=GET$:UNTIL A$="Y" OR A$="N"
1650 IF A$="N" SNAG=TRUE:ENDPROC
1660 PRINT''"Your score is ";STR$(score%)
1670 PRINT"Well done!"'"Would you like another game (Y/N)?"
1680 REPEAT
1690 A$=GET$:UNTIL A$="Y" OR A$="N"
1700 IF A$="N" END ELSE RUN
```

Explanations:

1610: If not going up, or player at level 2 or deeper (recall how we store room co-ordinates), ignore this procedure.

1620: Tell player, and seek advice.

1630-1640: Acquire a yes/no answer.

1650: If player chooses to remain, set SNAG as a signal to finish the main loop, and leave procedure.

1660-1670: Tell player the (good?) news about score, and seek advice again.

1680-1700: Note the lack of an ENDPROC statement. We end either with END, which is final, or by RUN, which is equally final!

So the player can move. Is there a room there? If not, we must make one. This procedure takes an argument K%. If K% is 1, we're calling the procedure normally. A lack of a possible room will trigger SNAG, etc. If K% is 2, however, we're calling the procedure from TRAP, and we don't want to set SNAG. Hence the 'signalling' to the procedure via K%.


```
1800 DEFPROCROOMTHERE(K%):LOCAL I%,J%
1810 IF dir%>2 I%=-1:J%=2*(5-dir%) ELSE I%=1:J%=2*dir%
1820 IF nroom%<90 ELSE SNAG=TRUE:IF K%=1 PROCSAYBLOCK:ENDPROC ELSE
ENDPROC
1830 check%=WHERE%(r%)+I%*10^J%:r1%=0
1840 FORI%=1 TO nroom%
1850 IF check%=WHERE%(I%) r1%=I%:I%=nroom%
1860 NEXT
1870 IF r1%>0 ENDPROC
1880 nroom%=nroom%+1:r1%=nroom%:WHERE%(r1%)=check%
1890 PROCMAKEROOM(r1%)
1900 ENDPROC
```

Explanations:

1810: This looks rather complicated. We have to work out what the co-ordinates of this new room will be, given the co-ordinates of the old room r%. Depending on the value of dir%, we have to change either I or J or K by the following formula:

dir%	change
0	J -> J + 1
1	I -> I + 1
2	K -> K + 1
3	K -> K - 1
4	I -> I - 1
5	J -> J - 1

(For example, going in direction 4 means going West, which decreases I by 1.) If you examine the list carefully, it has a pattern of sorts (which was why the order of exits was so arranged). Had you not thought of a pattern system, your program would have needed an ON dir% GOSUB. Don't worry. It wouldn't have mattered violently, and it would still have worked. Anyway, to work out the value of CHECK% for the (possibly new) room, we have to modify the value for the room r% which the player is in. We shall do this in line 1830, and it will involve adding or subtracting either 1, 100, or 10000 to the current CHECK%. We add for dir% = 0, 1 or 2; we subtract for dir% = 3, 4, or 5. The above table shows this clearly. So we set the sign of the addition to be plus (I% = +1) or minus (I% = -1) accordingly. The amount is 1 if dir% is 0 or 5; 100 if dir% is 1 or 4; and 10000 if dir% is 2 or 3. If we think of these as powers of 10 (i.e. 0, 2, or 4) we can see that J% is set accordingly also.

I stress again that this is just a fancy mathematical way of doing something. If you can't figure out a way to do something neatly when writing a program, it really isn't worth sleepless nights. Just program it slightly messily and get on with the rest of the program!

1820: If we've run out of rooms, set SNAG anyway. Tell the player about blockage in normal case, and leave procedure.

1830: Set new CHECK% in check% by adding appropriate term. In our example above of West (dir% = 4), I% would be -1, J% would be 4, so we would subtract 10000, correctly. Now set r1%, which will be the number of the new room if we find it, to be 0.

1840-1860: Search through the rooms we have already, looking for a CHECK% which matches the test value check%. If we find one, set r1% to the room number, and end the loop immediately by setting I% to the end value of the loop.

1870: If we found a room which matched, r1% will be nonzero, so quit.

1880: We didn't. Increase nroom% by one (it wasn't 90, because we checked earlier). Set r1% to this new value of nroom% – because it's a new room and hasn't a number as yet – and set CHECK% for this room to the value check% we calculated.

1890: Make a room numbered r1%.

The last line, of course, begged the question! But writing 'make a room' as a separate procedure allows the testing of PROCROOMTHERE. Running the program either throws up a bug so far, or reports the lack of PROCMAKEROOM (which you knew anyway). So don't be afraid to put off writing things for a bit. Here it is now:

```
2100 DEFPROCMAKEROOM(r%):LOCAL I%,J%,K%
2110 R%(r%)=(RND(2)-1)*1000000
2120 I%=RND(10)
2130 IF I%<3 R%(R%)=R%(r%)+110011:GOTO 2160
2140 IF  I%<7  R%(r%)=R%(r%)+110011:REPEAT  I%=RND(6)-1:  UNTIL I%<>2 AND
I%<>3:R%(r%)=R%(r%)-10^I%:GOTO2160
2150 REPEAT I%=RND(6)-1:UNTIL I%<>2 AND I%<>3:R%(r%)=R%(r%)+10^I%:REPEAT
J%=RND(6)-1:UNTIL J%<>2 AND J%<>3 AND J%<>I%:R%(r%)=R%(r%)+10^J%
2160 IF RND(6)=1 OR (dir%=3 AND RND(2)=1) R%(r%)=R%(r%)+100
2170 IF RND(6)=1 OR (dir%=2 AND RND(10)<4) R%(r%)=R%(r%)+1000
2180 IF R%(r%)<1000000 ENDPROC
2190 J%=WHERE%(r%)DIV10000:IF J%>4 J%=4
2200 CONTENTS%(r%)=0
2210 FOR I%=1 TO J%
2220 IF RND(2)=1 K%=FNDENIZEN ELSE K%=FNTREASURE
2230 CONTENTS%(r%)=CONTENTS%(r%)+K%*100^(I%-1)
2240 NEXT
2250 ENDPROC
```


Explanations:

2110: Give the room a 50% chance of being a cave or a passage. So set the millions digit with 50% probability. Notice that r% is local to this procedure, as it is handed through as a parameter.

2120: Choose a number from 1 to 10. We shall allow 20% of rooms to have all 4 horizontal exits; 40% to have 3 of the 4 possible horizontal exits; and 40% to have only 2 horizontal exits. The point is to make it not too easy to wander around, and to have a fair proportion of blockages. If you find the proportions not to your taste, change them here.

2130: I can't find a neat way to do this! There is a 20% probability of all 4 exits (110011 pattern). We add this to R%, which may already have a millions digit set. Jump to 2160 (a GOTO! Sorry.)

2140: 40% probability of 3 exits. Set all 4 as in 2130. Then scan I% through 0 to 5 randomly until it's not 'up' or 'down' (2 or 3). Then subtract that power of 10 from R% to remove that digit 1 from the pattern. Jump to 2160. Sometimes programs just are ugly. . .

2150: 40% probability of 2 exits. Randomly choose one with I%, and add it into R%. Choose another in J% (making sure it isn't the same as I%) and add that in too.

2160: A 1 in 6 chance of a down exit in the new room, unless player is going upwards, when it's 50%. We don't want it to be too difficult to change levels upwards (retreating).

2170: Similarly for an up exit in the new room, but only a 40% chance if the player is going down – making it more difficult to go down than up.

2180: If room is a passage, quit.

2190: It's a cave, so needs filling with J% things. J% is the level of the cave, but can't get bigger than 4.

2200: Set CONTENTS% to 0 initially.

2210-2240: Make J% random contents. Each one, whose number is K%, is equally likely to be a denizen (number FNDENIZEN, defined below) or a treasure (number FNTREASURE, ditto). Since we store two digits in CONTENTS%, raise 100 to the appropriate power (I% – 1). So the first thing goes in at digit 10 to the power 0, which is 1. The next at 100 to the power 1, which is 100, and so on. Then quit.

FNDENIZEN and FNTREASURE can be defined to suit yourself. Here are my options:

```
2300 DEFFNDENIZEN:LOCAL I%,J%,K%
```

```
2310 I%=RND(100):RESTORE 2340
2320 J%=0:REPEAT J%=J%+1:READ K%:UNTIL I%<K%
2330 =J%
2340 DATA 4,15,22,35,45,55,68,80,90,100
2400 DEFFNTREASURE:LOCAL I%,J%,K%
2410 I%=RND(100):RESTORE 2440
2420 J%=10:REPEAT J%=J%+1:READ K%:UNTIL I%<K%
2430 =J%
2440 DATA 35,55,75,95,100
```

Explanations:

2310: Choose a random percentage I%.

2320-2340: 3% chance of a dragon (value 1) else 11% chance of a giant (value 2) else . . . else 11% chance of a lion (value 10). Just change these numbers if you prefer others. More dragons to fight? Then 2340 begins DATA 9, etc.

2410-2440: Similarly for treasure. Make bronze likely (34%) and a treasure chest (value 15) only 6%.

2.6 Programming – the procedures (2)

Most of the work is now behind us, although the program remains untestable. If you're typing this in as we go, load a collection of 'dummy' procedures and let the program run. If you've never used them, a 'dummy' procedure is something like 'DEFPROCHELLO: ENDPROC', inserted specifically to allow testing of other parts of your program. Now, on with the job. We first need to check whether the player can enter the new room.

```
2500 DEFPROCENTERROOM
2510 SNAG=NOT FNEXIT(5-dir%,r1%)
2520 IF SNAG PROCSAYBLOCK
2530 ENDPROC
```

Explanations:

2510: Player is leaving his old room in direction dir%. This means he is entering his new room by direction (5 – dir%). This finally explains why the exits were set up in that order – purely to make this line be an easy calculation! Again, if you hadn't thought of something along those lines, and had done 6 IF statements, it really wouldn't have mattered, except that the habit of 'thinking of neat ways' is a useful one to acquire. So SNAG gets set if there's no exit in room r1%.

2520: If so, say the exit is blocked, and quit.

To do this, we use:


```
2000 DEFPROCSAYBLOCK
2010 PRINT'!"That exit seems to be blocked!"
2020 PROCDELAY
2030 ENDPROC
```

which needs no explanation. The next item is to check for earthquakes and traps:

```
2600 DEFPROCEARTHQUAKE
2610 IF RND(25)>1 ENDPROC
2620 PRINT'!"An earthquake just reduced the area"!"you were in to
rubble!"
2630 PROCDELAY
2640 SNAG=TRUE:R%(r%)=0:RETREAT=FALSE
2650 ENDPROC

2700 DEFPROCTRAP:LOCAL I%
2710 IF RND(25)>1 ENDPROC
2720 dir%=2:PROCROOMTHERE(2)
2730 IF SNAG ENDPROC
2740 PRINT'!"You fell into a trap!"!"Your party just dropped a level!"
2750 RETREAT=FALSE:PROCDELAY
2760 ENDPROC
```

Explanations:

2610: A 1 in 25 chance of an earthquake.

2620-2640: Tell player, let him read it, use SNAG to signal, reduce room to rubble (R%(r%) = 0 does that very efficiently) and remove retreat.

2710: As 2610.

2720: Reset direction of movement to be down (the player is falling!) Make a new room below but use option 2 now (don't say things about blockages)

2730: If we can't make a room (all 90 used up) he can't fall down the trap!

2740: Yes he can, so tell him.

2750: No retreat, and let him read 2740.

We now skip through to line 210 of the main program loop, where we are examining retreat possibilities. We called up a function FNANYDENIZENS on that line, which returned TRUE or FALSE as there were, or were not, any denizens present.

```
2800 DEFFNANYDENIZENS: LOCAL I%,J%,K%
2810 J%=FALSE
2820 FOR I%=0 TO 6 STEP 2
2830 K%=FNWHO(r%,I%)
2840 IF K%>0 AND K%<11 J%=TRUE:I%=6
2850 NEXT
2860 =J%
```

Explanations:

2810: The function will take the value J%. Set it to FALSE unless we find a denizen in the room.

2820: Loop through the 4 possible things in room r%.

2830: This one is K%.

2840: If K% has a value consistent with denizens, set J% to TRUE, and kill the I% loop by making it 6. DON'T jump out of the loop by writing 'IF K%>0 AND K%<11=TRUE', as this leaves an unfinished loop.

2860: Return the value J%, whatever it is by now.

A little later we need PROCEMPTYROOM:

```
2900 DEF PROCEMPTYROOM:LOCAL I%,J%,K%
2910 J%=FALSE
2920 FOR I%=0 TO 6 STEP 2
2930 K%=FNWHO(r%,I%)
2940 IF K%>10 score%=score%+CHAR%(K%):J%=TRUE
2950 NEXT
2960 CONTENTS%(r%)=0:SAFE=TRUE
2970 IF J% PRINT'"Your party removes the treasure!":PROCDELAY:PROCDELAY
2980 ENDPROC
```

Explanations:

2910: There may be treasure present. Set a marker to say whether there was. Assume not, unless proved otherwise.

2920: Check the 4 things in the room.

2930: Get the number of one.

2940: If treasure, add its value to score%, and note treasure for J%.

2960: Empty the room and set SAFE.

2970: If there was treasure, tell player. Give him a bit longer than usual to take all this in.

Now the interactions with the denizens must be added. Your program will work fine now, by the way, until you find a room with denizens! The first procedure is approaching.

```
3000 DEF PROCAPPROACH:LOCAL I%,J%,K%,attack%,friend%
3010 K%=10:FOR I%=0 TO 6 STEP 2
3020 J%=FNWHO(r%,I%):IFJ%>0 AND J%<11 IF J%<K% K%=J%
3030 NEXT
3040 attack%=(CHAR%(K%) DIV 100) MOD 100:friend%=(CHAR%(K%) DIV 10000)
3050 I%=RND(6):NEUTRAL=FALSE:IF I%<= attack% SNAG=TRUE:bonus%=-1:
PROCDELAY:PRINT'"The denizens attack you with surprise!":
PROCDELAY:ENDPROC
3060 IF I% < friend% NEUTRAL=TRUE:FIGHTING=FALSE:PROCDELAY:PRINT'"The
denizens ignore you!":PROCDELAY:ENDPROC
3070 FOR I%=0 TO 6 STEP 2
3080 J%=FNWHO(r%,I%):IF J%>0 AND J%<11 AND nteam%<30
nteam%=nteam%+1:TEAM%(nteam%)=J%
3090 NEXT:PROCDELAY:PRINT'"Your approach is successful!"'"The denizens
join your party":PROCDELAY:FIGHTING=FALSE:SAFE=TRUE:PROCEMPTYROOM:ENDPROC
```

Explanations:

3000: We don't really need attack% or friend% to be

local – we could use L% and M%. It's just easier to remember what they are here.

3010: Prepare to find the lowest numbered character in the room (there must be one else we wouldn't be here). Use K% for this, and set it to 10 (the highest possible) first. Scan through 4 things with I%.

3020: This thing is numbered J%. If it's a denizen and less than K%, reset K% to its value.

3040: Compute the attack and befriending thresholds for denizen number K%. The extraction method should now be straightforward.

3050: Throw a 6-sided die (I%). Set NEUTRAL to be FALSE for now. If the denizens attack, set the trusty SNAG. Set the bonus to them (all bonuses will be added to player, so setting bonus% negative is the same as adding to the denizen's forthcoming score. This is a very useful trick if you've never come across it). Tell the player what's happening to him, and quit.

3060: If denizens neutral, reset the marker, clear FIGHTING, say so, and quit.

3070: They befriend! Count through the 4 possible things in the room.

3080: If J% is a denizen and our team isn't full, add one to nteam% and insert J% in the TEAM% array.

3090: Tell the player, reset logical values, empty the room, and quit.

We have put it off to the end, but must now write the fighting procedure. Fortunately the hard work is all done for us already.

```
3200 DEFPROCFIGHT:LOCAL I%,J%
3210 you%=bonus%:them%=0:FIGHTING=TRUE:J%=nteam%:IF J%>3 J%=3
3220 FOR I%=1 TO J%
3230 you%=you%+(CHAR%(TEAM%(I%)) MOD 100)
3240 NEXT
3250 FOR I%=0 TO 6 STEP 2
3260 J%=FNWHO(r%,I%)
3270 IF J%>0 AND J%<11 them%=them%+(CHAR%(J%) MOD 100)
3280 NEXT
3290 you%=you%+RND(6):them%=them%+RND(6)
3300 IF you%=them% PRINT'"You're still
fighting!":PROCDELAY:bonus%=0:ENDPROC
3310 IF you%<them% PROCLOSE ELSE PROCWIN
3320 IF NOT FNANYDENIZENS SAFE=TRUE:FIGHTING=FALSE:PRINT'"You killed all
the denizens!":PROCDELAY:PROCEMPTYROOM
3330 ENDPROC
```

Explanations:

3210: Set the fighting scores for the player (you%, set to the bonus so we don't forget it) and the denizens (them%). Set FIGHTING, of course. Allow a maximum of 3 of player's team in the action.

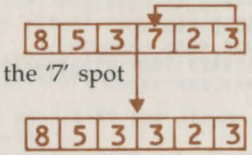
3220-3240: Add to player's fighting score the fighting strength of the first J% of his team.
3250-3280: Add to denizens' fighting score the fighting strength of all of them.
3290: Add a random dice throw to both scores.
3300: If equal, say so, remove any surprise bonus, and quit.
3310: Either player wins or loses.
3320: If he's killed all the denizens, it's safe and the fighting has ended. Tell the player, and pick up any treasure.

Which only leaves two procedures, for winning and losing a fight:

```
3400 DEFPROCLOSE:LOCAL I%
3410 IF nteam%>1 ELSE PROCDELAY:PRINT''"Your party is all dead!"''"Would
you like  another  game?":REPEAT A$=GET$:UNTIL A$="Y" OR A$="N": IF A$="Y"
THEN RUN ELSE END
3420 I%=RND(nteam%):PROCDELAY::PRINT''"The denizens killed"'
CHR$(TEAM%(I%)): PROCDELAY
3430 TEAM%(I%)=TEAM%(nteam%):nteam%=nteam%-1
3440 ENDPROC

3500 DEFPROCWIN:LOCAL I%
3510 I%=-2
3520 REPEAT I%=I%+2
3530 J%=FNWHO(r%,I%)
3540 UNTIL J%>0 AND J%<11
3550 PROCDELAY:PRINT''"Your party killed"'CHR$(J%):PROCDELAY
3560 CONTENTS%(r%)=CONTENTS%(r%)-J%*10^I%
3570 ENDPROC
```

Explanations:
3410: No team left? Then give player the option of a new game.
3420: We have to kill one member of the team, number I%. Say which member it was.
3430: We now delete that member. A neat way to remove it from the TEAM% list is to set the I%th member of the list equal to the nteam%th (i.e. the last) member, then drop nteam% by one. What happens looks like this:



Copy the 3 at the far end into the '7' spot

(note that the last 3, although stored, doesn't really 'exist' any more.

3510-3520: Loop I% by 2's.

3530-3540: Until we find a denizen J%.
3550: Tell player he killed it.
3560: Remove J% from its position in CONTENTS%.

Well, that's the entire game. Try playing it for a while – it isn't easy to get a large score and to live to tell the tale!

2.7 Improvements

The previous sections have described what is still the bare bones of a 'hack-and-slash' game. You will probably want to make some changes or improvements to it after you have played it. Doing so will be excellent practice in the care and handling of databases! In this section a few modifications are suggested, which you could try out as exercises.

(a) *Increasing the number of different denizens and treasures.* This is not a difficult job. Apart from a few dimension and data statements, it's only a matter of checking through the procedures and modifying where necessary. Had we left the number of denizens as a variable (e.g. ndenizen%) the checking wouldn't have been necessary, as all the arithmetic would have worked out properly once the variable was set in PROCSETUP. This points to a general programming principle: always use variables rather than specific values like 10 (because you will have forgotten where all the references to that 10 were after a few weeks!)

(b) *Modifying the chances of survival.* This, too, is straightforward. You could modify strengths and thresholds immediately, or make the probability of more or less exits be whatever you wished.

(c) *Modifying the fighting procedure.* The fighting is – deliberately – the least 'realistic' procedure in the game. You could, instead, allow your team to square off against the nasty denizens, with doubling up if you have a big team. Or borrow a fight procedure from a role-playing game and allow all sorts of actions like ducking, blocking, firing arrows from long distance, etc. There are numerous options.

(d) *Modifying the properties of treasure.* All the treasure in the game is essentially passive, in the sense that it disappears the moment it's picked up. You could alter this in at least three ways. First, allot each piece of

treasure to a specific member of the party – which you would do in PROCEMPTYROOM – and lose any treasure on a member who gets killed. This ensures the care and treatment of treasure, especially when combined with option (c) above. Second, you could allocate different weights to different treasures, and give each member of the party a different carrying ability. This would force the player to approach denizens rather more frequently, as he will run out of carrying ability fairly early. Third, certain treasures could be magical. For example, incorporate a magic ring able to restore to life one party member – once. Handling this would require remembering which member just died, and a logical flag noting the possibility of resuscitation.

(e) *Improvement of room descriptions.* Areas in this game are either caves or passages. It would be more attractive, and certainly more realistic, to add some sort of description to some of the areas ("A cobweb-filled passage", or "A volcanic dungeon", etc.). Recall all those unused digits in R%? We only used a 1 or a 0 to trigger the descriptions, but you could do far more than that with a separate array to hold more elaborate ones. (Note that we cannot simply make them up in a random fashion whenever the player enters an area. The information must be stored since he may return to the area and would be confused if its description has changed in his absence.)

These are no more than suggestions. Let your imagination run wild, and modify the program as you see fit. Your only limitations are space, a problem to be looked at later in the book.

3

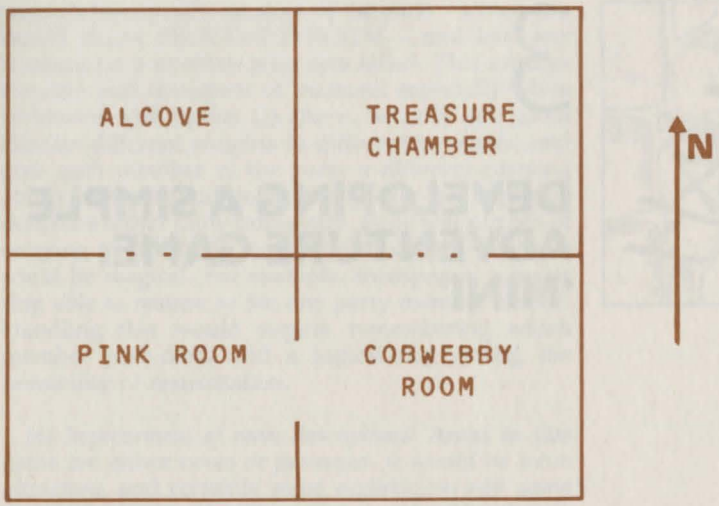
DEVELOPING A SIMPLE ADVENTURE GAME: 'MINI'

In this Part we turn (finally) to the writing of a genuine Adventure game. If you have the game on the associated cassette (available separately from Acornsoft), try it out before the plot is revealed. The object is simply to get the crown before drowning.

I have constructed a simple plot to demonstrate some desirable and some of the less desirable features of a typical Adventure game. We shall design a method for handling the plot that reveals little that is new but probably contains all that is required for writing simple games. In the next Part we shall see that there are better ways to do most of the things I describe here, but it needs a simple game to make their need apparent.

3.1 The plot

First, let's look at the map. There are only four rooms, and exits lead only along the four main compass directions. This removes NE, SE, SW, NW, up and down, but as we saw in Part 2 this merely shortens things, it doesn't reduce the complexity. The player is 'born' in the cobwebby room.



A gap indicates an exit. There's only one ordinary one, from the pink room to the cobwebby room and vice versa. The other two exits, marked with a couple of dots, are special because something unusual may happen to the player when he tries to pass through them. The exit to the treasure chamber (which is where the player is trying to reach) is blocked by a locked door. It'll have to be opened or removed before the exit can be passed. The other 'trick' exit, between the pink room and the alcove, contains whirling knife blades which threaten to cut the player to ribbons if he goes through them.

We shall give the player some tools. In addition to the jewelled crown in the treasure room (which could be a tool, but isn't in this game!) there is a shield in the cobwebby room, a black rod (a popular feature of Adventure games) in the alcove, and a helpful magic word 'BLAH' written on the wall in the pink room.

These tools function as follows. The shield offers protection against the knives, but only if the player is actually wearing it (rather than just carrying it) when he passes through. If he survives, the shield will have shattered, so he'll have to figure out another way out

of the alcove. This is provided by saying BLAH, which flies him back to the cobwebby room, his starting point. Finally, waving the rod in front of the door causes the door to vanish.

To make life suitably worrying, there's a time limit for the player, as water is rising steadily throughout the game, and he will drown if the crown isn't picked up before the passing of a certain number of turns.

So that's the plot: three problems to solve, and a certain time in which to do it. But before we consider how we can program it efficiently, let's pause for a moment and look at the plot as a collection of puzzles. Certainly the puzzles are not in themselves earth-shattering, but the structure of them is of interest. The solution to the locked door (waving the rod) is the first problem the player meets on entry to the game. However, because of the game structure, it's the last problem he can actually solve. Something along these lines is important in Adventure games – some problems ought to be fairly easy to solve; but others should be designed to test the player for some considerable time. When, finally, he can exclaim, 'Ah, so that's the solution!' a player has derived pleasure from playing the game (which is the point of playing it, after all).

Consider the magic word. Almost the first thing anybody will, or ought to, do when seeing it is to try saying it. So, it's important that it be put where it doesn't work otherwise you'll lose a problem immediately. Mind you, the poor player shouldn't really have to run around 200-odd rooms muttering 'BLAH' in every room just to find where it does something. Ideally, as here, the possibility of a magic word being useful should occur to him only in certain correct circumstances. At every point, try to channel the player's thoughts in the direction you'd like them to go: hints are often useful!

Then there are the whirling knives. It is vital that the player get a different response when he goes through the exit without the shield, than when he goes through carrying, but not wearing, the shield. He'll be killed in either case, but he must receive a hint (i.e. a different response) that he's in some sense 'close' to the correct answer. So we must remember to refer to the shield as we kill him!

Finally, there's the mounting water problem. If we were being nasty, we could calculate the shortest number of moves necessary to solve the problems and give the player exactly that number of moves. I don't subscribe to this; I believe that players should (unless the structure of a given puzzle makes this impossible) have a bit of leeway in their timing. Enough, say, to permit the occasional 'LOOK' to remind themselves of where the exits are! But don't give them too long to solve the game – else the water will cease to be a problem. The fun of the game lies in worrying the player, but not unduly so.

To give you a feel for the flavour of 'MINI', here's a little specimen dialogue from the game:

You are in a crumbling room full of cobwebs. A passage leads west. A barred door bars a north exit. You can see a shield. . .

GET SHIELD OK There are faint sounds of water.

GO NORTH A barred door bars a north exit. The sounds are slightly nearer.

W The water sounds are quite loud. You are in a cheerful pink room. The word BLAH is inscribed on the ceiling. A passage leads east, and a doorway full of whirling knives leads north. You can see: Nothing

NORT The knives stab at you as you pass. They slice you to ribbons. You have departed this world, alas. Would you like another game?

This Part will follow the plan of Part 2. I intend to spend some time discussing the problems facing us, in English, before we begin to program in BASIC. Understanding general principles is more important than working out an individual program.

3.2 More ideas about databases

The 'MINI' game will extend the database ideas already discussed, and introduce some new ones. At the same time, many of the apparently complicated ways in which we handled storage in Part 2 (all those digits, MOD 10's, etc.) will disappear – not because they weren't useful, but because there are other ideas to examine here. In Part 4 the best of all our ideas will come together.

Let's start with the rooms section of the database. As before, we shall have to store a list of exits, and their destinations, for each room in the game. Unlike 'CAVES', however, we know the map before the game starts, so we can store the room exits and destinations together. One of the methods discussed in Part 1 will suffice.

But how can we handle the location of the objects? In 'CAVES', the objects 'vanished' upon being picked up, merely becoming part of the score. (This was necessary because no room could hold more than four things, you'll remember). In this case, there is no restriction – if the player wants to pile up all the objects in one room, he's at liberty to do so (provided that it is possible in game terms!). But this makes storing a list of objects for each room awkward as we don't know how long the list should be. We could invert the method we used in 'CAVES' and use an array of 'object rooms', one entry per object. If we call this array or%, ('o' for object, 'r' for room) then or%(3) is the room that object 3 is currently in. So, to know which room an object is in, we merely look it up in the array. Conversely, to know what's in a given room, we scan through all the objects and jot down which ones live in which room.

But what room do we allocate to objects carried by the player? At first sight you might suggest that we put them in the actual room the player is in ('telling the truth'!). A little thought shows why this won't work. We would have no immediate way, for example, of distinguishing objects *carried* by a player in room 3 from those that are *set* in room 3. And if the player then moves to room 4, how do we know which of the objects purporting to be in room 3 should move with him?

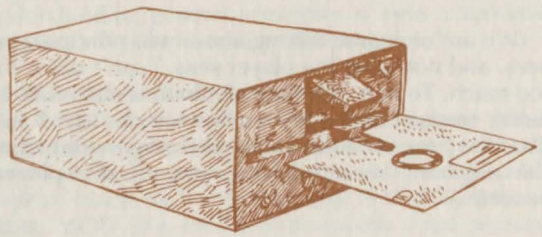
(We are of course talking about what the program sees, and not what the player sees. I can't stress this too much. To him, the dead elephant is obviously too heavy to carry and will remain where he saw it last. That the program has to struggle to figure out where the elephant should be is none of the player's concern.)

So where do we put objects carried by the player? One method would be to generate a second array for the objects, each entry carrying a flag to say 'carried' or

'not carried'. (This is ugly, and uses up too much store.) A second method is to make the room values negative if the object is being carried. (Less ugly, as it uses only the one array, this method is still no good for direct byte storage to be explained in Part 4.) The simplest solution, and one which is used by many games on the market, is the one we shall use in this book. (Its disadvantage is that it isn't totally flexible, but in problem circumstances it is usually easy to 'fake'.)

The method is straightforward. We create an extra room devoted solely to objects carried by the player. It isn't really a room in game terms, but it gives us somewhere sensible in the machine for the objects to reside. The player can then roam as he wishes. The objects remain in this special 'room', which we shall number as room 1, until he drops or loses them. When we want to display what objects the player is carrying, we look through the objects and find those for which or% has the value 1.

While we've been discussing this setup, something subtle has occurred. The database has developed two distinct features. One part of the database always remains the same, no matter how many adventures the player has (where each exit leads, for example). But the other part of the database is constantly changing (which objects are in which rooms, etc.). You may not feel that this distinction is significant; nor is it for this game. But suppose we permitted the saving of a game onto tape or disc while it was still being played. On disc, where access times are so fast, the size of the database won't really affect the process. But on tape, the faster the loading and saving of a game, the better.



The point is that by distinguishing between the unchanging, or static part of the database and the

storyline (as it were), saving on tape mid-game can be greatly facilitated. There is no need to dump out the 'backdrop' (the static part of the database) because that is the same for all games. Only the 'storyline' (the dynamic part of the database) has to be copied out. So when we design a larger game, we should give some thought to the whereabouts of both static and dynamic parts of the databases live.

3.3 Yet more ideas about databases: states and exit programs

When we were setting up the plot, several of the actions a player might take to try to solve one of the puzzles depended on the history of the game – i.e. what had happened so far while he was playing. For example, was he wearing the shield? (So two things had to have happened to the shield: it had to have been picked up and, therefore, be in the player's possession, and also it had to have been 'worn', however we code that.) Again, whether he could go north into the treasure chamber depended on whether he had removed the door!

You may be tempted to invent a couple of variables to look after this: WEAR, which can be TRUE or FALSE depending on whether the shield is being worn, and NODOOR, which could also be TRUE or FALSE depending on the state of the door. Most books – and programs, as far as I know – do just this. Obviously it is exceedingly efficient for the small game we're discussing here. But in a complicated game with about 60 objects, 80 rooms, etc., there might be too many such 'extra' variables to keep track of. Still no problem, you may think, albeit a little messy. But we can do things a lot more cleanly, and more efficiently.

Let me introduce you, then, to the concept of STATES of rooms and objects. We shall define two arrays, one for rooms and one for objects. The room array has an entry for each room (and similarly the object array has an entry for each object), which contains a single number allocated to that room. The value of this number will tell us (and the program) what has been happening to that room. Of course, this is a bit like overkill for the little game here, but you'll be able to see its general usefulness as we go along.

How will this work? Consider the cobweb room

with the north door. We'll set the state of all rooms to zero to start with, lacking any good reason to do otherwise. When the player waves the rod (which you will recall makes doors vanish) in the cobweb room, we'll set the state of the cobweb room to 1. Whenever a player tries to leave a room, we offer its state for examination and the program acts accordingly.

The shield is handled in much the same way. Its state is zero to begin with, and 1 when the player says 'WEAR SHIELD', assuming he was holding it. (We must also remember that if he drops it, its state will revert to zero. Thus, when he picks it up again, the program does not presume that he's wearing it!) Then, when he goes past the knives, our action depends, amongst other things, on the state of the shield.

Two arrays of states may seem more cumbersome than the 'simple' solution, but the beauty of the system will become apparent in Part 4 when states are seen to permit automatic descriptions of rooms and objects, without lots of ifs and buts. As a hint of things to come, however, note that states allow you to do familiar things, like filling an empty bottle with water, very easily. (The database we have been using can only let the player carry objects. How can a bottle 'carry' something called 'water'? With states, the problem becomes trivial. A bottle in state zero is empty; in state 1 it contains water!)

As far as the program is concerned, there are no objects called doors or knives. There are merely things that happen to the player as he passes through an exit. Look again at the logic of going north through that door. What the player 'sees' is a door. What the program sees, however, is a hiatus after the player types 'N' or 'NORTH' or 'GO N', or whatever is the appropriate instruction. During this hiatus the program runs to a subprogram which examines the state of the cobweb room and decrees whether or not the room may be left. A 'door' simply doesn't come into it!

What of the knives? When the player says 'N' in the pink room, once again a subprogram is scanned: tell the player about the knives; is he carrying the shield? If not, kill him. Is he wearing the shield? If not, kill him, but drop him a hint as to why he has been killed. Otherwise, destroy the shield and let him pass

through. Again, note that there is no mention of an object called 'knives'. They don't exist.

These non-existent objects are replaced in program terms by subprograms tied to specific exits. Before the player may pass through an exit like this, he must first execute the particular subprogram and suffer the consequences. We call these subprograms 'exit programs', for the sake of a bit of jargon. The concept is a powerful one; my own games typically contain 30 or more exit programs, many of which the player is never aware! Their use is far more widespread than exemplified here, and we'll see more examples in Part 4. But here's one type of exit program to whet your appetite. Suppose you wanted to create a random maze, which would be different with each game but not change within a specific game. What you could do is set up an exit program two rooms before the maze, which runs something like this: is the state of the next room 1? If so, leave the exit program. Otherwise: set up a random maze, and set the state of the next room to 1, then leave the exit program. Notice that the player remains ignorant of all this! The maze is set once, and once only, by this subprogram.

3.4 Messages

Telling the player what is happening to him merely involves a PRINT statement in BASIC. Anything that the player reads is a MESSAGE given him by the program. For example, LOOK is the standard code for 'give me a description of the room I'm in'. The program's response will take the form of a message describing the room itself; then, under normal circumstances, there follows a whole list of messages, each one describing an object present in the player's room. Similarly, when the player gets killed going through the knives, he will receive four messages. First comes one about the knives themselves (he'd get that no matter where the shield was) and then a message telling him of his demise. This would then be followed by a standard 'Oh dear, you're dead' style message, and this in turn is followed by another, asking about a new game.

Now the point of all this is that you should train yourself into thinking about all communications as messages, rather than thinking: when the player gets the knives, I must PRINT 'The knives slice you to pieces.' There are some good reasons for this.

First, consider that task of describing the player's room. Assuming R% to be the value of the player's room, isn't it rather ugly, not to say inefficient, to write 'IF R%=2 PRINT "You are in a cobwebby. . .." ELSE IF R%=3 PRINT ". . ."'? To follow this, you'd have to loop through the objects, and decide which object descriptions to print also. Obviously there are better ways to do this!

Instead you can store ALL messages (barring a few odd ones like 'I don't understand!') in the static part of the database. Each message will be given a unique identifying number. So message 13 might be – and is in this game – 'There are faint sounds of water.' On the first occasion that we tell the player about the water rising, we can now write 'PROCm(13)', which will mean 'print message 13', instead of 'PRINT "There are faint sounds of water."'

Apart from the obvious neatness (we don't disturb the flow of program logic when we talk to the player), this method of storing messages almost fully automates descriptions of rooms and objects. With it, we can store a message number for each room and each object and simply PROCm it when necessary.

I say almost fully automated because there are still some ifs and buts. What of the door, which may or may not be there? There will have to be a message associated with it, which will be printed as part of a description of the cobwebby room, provided that it's in state zero. Similarly, the shield, when the player does an INV(entory) of his possessions, will have to have an extra message "(which you are wearing)" tacked on if the shield is in state 1.

Apparently, we should be stuck with this kind of awkwardness no matter how we handled descriptions. I hope you find it unaesthetic, because I do. In Part 4 we'll meet a way of removing all the messiness by modifying the way we treat and store messages.

3.5 Vocabulary

Now let's consider how the vocabulary of Adventure games is made up. In this context, 'vocabulary' means those words that the player may type which the computer can recognise. There are several classes of such words.

The obvious class is VERBS. These include DROP, LOOK, THROW, and so on. We anticipate finding verbs as the first thing a player says to the program (as in 'DROP ROD', 'GET SHIELD', etc.). This isn't to say we'll always find them in that position! Certainly, any player who types in 'ROD DROP' deserves an 'I don't understand that!' message in response. But in Adventures not all 'doing' words are in fact verbs. In 'GO NORTH', 'GO' is obviously a 'doing' word, but in many games, so is NORTH, or N. (Again, most players discover that GO or MOVE are an unnecessary waste of time; the original Colossal Cave adventure makes a point of reminding us of it!) But the player should still be entitled to say 'GO NORTH' if he wishes. So your program should be able to accept some non-verbs as commands.

What kinds of second words are most common? LOOK, INV (or INVENTORY) require none, of course, but the most common second words are THINGS – so named to keep clear in your mind the distinction between words the player can type, and program ingredients. THINGS are nouns that refer to objects, namely quantities which the program manipulates (such as the shield, the rod, and so on). Different THINGS can refer to the same object in the program if we choose to allow it. The jewelled crown might be called 'JEWEL' or 'CROWN' by the player – if we permit it. So this is an example of two THINGS referring to the same object (the crown).

We now know that there are also things that the player sees which the program doesn't. The knives and the door, in our game. What happens if the player says 'GET KNIVES'? We cannot include the word 'KNIVES' in our list of things because although in English grammar it is a noun, in our program in the strictest sense, the word does not refer to anything. There are two ways out of this dilemma. One is to shrug and let the player get 'I don't understand that!' when he tries 'GET KNIVES'. This seems grossly unfair to me: the room description included knives, yet the program will not tolerate a reference to them! The second way, which we shall espouse here, is to create another category of second words. These I shall refer to as SPECIAL words. They are special because they don't refer to what the program recognises as objects, yet the program must be able to comprehend the word.

In our small game, the only two special words will be DOOR and KNIVES; neither will ever succeed in getting a positive response from the program, but at least they'll be recognised!

Unfortunately, there are yet more types of second words. Suppose the player says 'SAY HELLO' to the program. A smart response – and fairly standard – is 'OK, "HELLO!"'. But notice that HELLO isn't on any of our lists at all; it could be any word at all. So either we must be prepared not to accept SAY HELLO, or else we must somehow treat any word after SAY as being acceptable.

It's very program-consuming to include in each command subprogram a decision as to what words are acceptable as second words. It's also not very neat. So, given that we have to store the vocabulary in the static database, let's store an extra quantity with each verb, to indicate what kind of second word can follow it. For most purposes, and certainly those here, there are three kinds of verbs:

TYPE 0 (Those which cannot be followed by a second word)
TYPE 1 (Those which must be followed by a thing or a special word)
TYPE 2 (Those which can be followed by anything or nothing)

Examples of type 0 would be EAST, BLAH, LOOK. Examples of type 1 would be GET, THROW, WEAR. And examples of type 2 would be GO, SAY, etc. There's no particular reason for my numbering.

So each 'doing' word (in our terms, each verb) will be stored with an integer (0, 1, or 2) to mark what kind of construction the program expects. Then when we are examining what the player says, we can throw out a lot of nonsensical input. But we must also store another integer which tells the program what command the verb intended. (The program can then be guided to a suitable subprogram by the value of this number.) It follows that by giving certain verbs the same identifying number, we can permit the use of synonyms (e.g. GET as well as TAKE),.

Having decided that verbs need two numbers, what of things and special words? Since these are rather less

complicated, they only need a single number with them. For things, the number refers to the object named by the thing. For special words, we can identify them in any way convenient for our program; it doesn't matter for our small game, since neither achieves a useful response. I'll just give DOOR a 1 and KNIVES a 2 and then ignore them.

3.6 Program structure

There is still one more item to discuss before we get down to writing 'MINI', namely the structure of the overall running program. If we can get that fairly clear in our minds, even in pseudo-code format, we're more likely to get an approximation to a working program on the first trial.

So here we go. I'm going to put approximate line numbers in as we go along to make comparison with the program somewhat easier later. Explanations will be given after the pseudo-code.

```
  10 –  99: Initialise variables, etc.
 100 – 199: Between turns housekeeping; describe
            room if moved.
 200 – 299: Input from player, isolate what he said.
 300 – 399: Check if command makes sense; if not,
            back to 200.
 400 – 499: Unused here (reserved for later!).
 500 – 599: Handle command. Die if necessary.
 600 – 699: Post-program routine (see below). Go
            back to 100.
1000 – 1099: Death and restart routine.
2000 – 2999: Command subprograms.
3000 – 3999: Vocabulary data list.
4000 – 4999: Static database: room and object data,
             and messages.
5000 – 5999: Helpful procedures (room and object
             describing, message printing,
             vocabulary locating, etc.).
6000 – 6999: Reserved for later.
```

Lines 10 to at worst 99 will set up initial values of variables, dimension arrays, etc.; the kind of thing one does at the beginning of most programs. In Part 4 we'll see how to avoid much of this. Lines 100 to 199 will usually only be a trivial test:

Check player's last room. If same as this, continue, else describe this room and reset the 'last room'

variable.

The idea is almost to automate when describing a player's room. Under all except unusual circumstances, we do this only after the player has moved. By keeping track of the last room as well as the current one, this is easy to detect. To ensure a description at the beginning of the game, we can set the last room to zero, so the program thinks the player has moved.

Lines 200 to 299 request player input, and must sort it into at most two words. I truncate all words to four letters for two reasons. First, it saves unnecessary typing by the player. Second, it facilitates quick movement by the pling operator (to be discussed in Part 4). Anyway, at the end of 299, the program should have two strings x$ and y$ which hold the first and second word typed, respectively.

Lines 300 to 399 attempt to locate the verb (x$) amongst the vocabulary, and should complain if they can't find it. Assuming all is well, they then check on the second word, bearing in mind the coding on the verb already (i.e. if it's a zero, then complain if there's a second word, and so on). By the end of this section the program should know what it's doing!

Lines 400 to 499 don't exist here. The use I put them to will turn up next Part.

Lines 500 to 599 actually do what the player told the program to do. There will be a massive ON C% GOSUB statement, where C% will hold the current command number, and all the GOSUBS are in 2000 to 2999 inclusive. On return from the appropriate GOSUB, a flag (F%) will be checked to see if one of several possibilities has occurred which need special treatment. These are:

Has the player died? If so, off to death program
Was the command one of the 'second word is a verb also' type?
If so, treat y$ as a new x$ and try again.
Was the command one to stop the program? If so, off to new game program

Lines 600-699 contain something new, which is called here the post-program. Instead of simply programming 'that's finished that command, now

back to line 100 for the next one', which will often suffice, we have added an extra piece of possible program. This is executed after the player's command has been acted on, but before we return to the room description possibility at line 100. Since it occurs after the main command program, we term it a post-program. (If you wonder about a pre-program, you'll see that in Part 7.)

What do we use a post-program for? In Adventure games, practically anything! For example, keeping track of the nasty things happening to the player. The rising water is a good example. By now you'll have realised that 'water' isn't an object in the game (and in fact isn't even in the vocabulary) but that messages about rising water will continue to appear on the screen. It's the function of the post-program to supply these messages, since they are in no way part of the response to the player's commands themselves.

We also have to finish the game when the player picks up the crown. We could program this into the 'take-prog', yet to be written. It's a little tricky to do that, but we could. It's infinitely cleaner and simpler to put the test into the post-program. We can enquire whether the player is holding the crown, at the end of every turn. If he is, then we congratulate him and end the game.

Post-programs have many other uses, too. If there are objects to be moved around, like monsters, they can be moved here. This will mean that when a room description is given next, assuming the player has moved, the description of the monster will be given correctly. However, we must be careful to end the post-program with an UNTIL FALSE to complete the command loop, else control will pass to the next set of lines.

These are lines 1000-1099, usually only two lines long, which include two procedures: a death program, which prints a 'You're dead' message, followed by a new game procedure, which prints 'Would you like another game?' and acts accordingly.

Lines 2000-2999 I reserve for the various command subprograms. These are usually written in whatever numerical order seems convenient; this tends to be alphabetical except for synonyms, as the verbs have to

be alphabetised anyway.

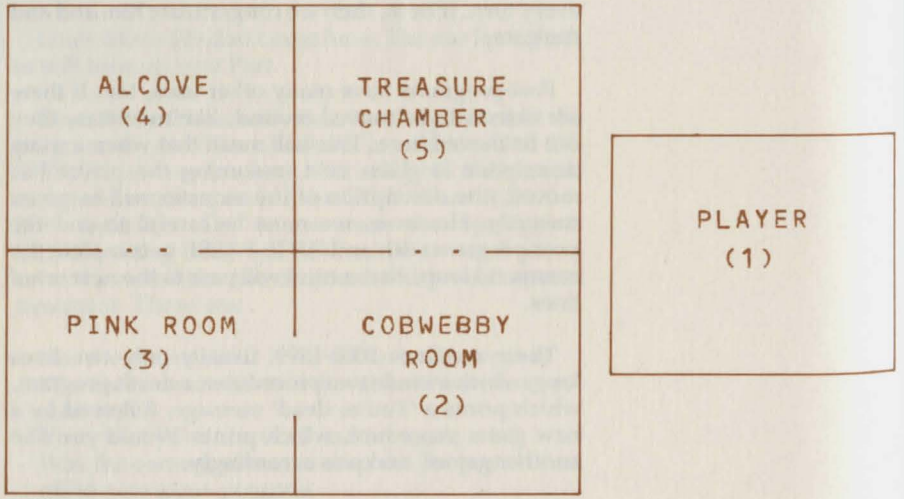
The remainder of the lines are self-explanatory. So let's program 'MINI'!

3.7 Writing the program (1) – objects and rooms

When I write an Adventure program I tend to use three or four pieces of paper simultaneously; hence my appeal to clear-headedness and organisational ability! (To follow my logic, a few pieces of paper certainly wouldn't come amiss.)

We'll need one sheet for the numbers part of the database, and one for the messages (we will have to refer constantly to that one). We also need one for the program as we write it.

The first thing I write (bottom-up programming, this) are the definitions of the objects and rooms. Let's start with the rooms. If you look back at the map, you'll see we have 4 rooms. In addition, there is the room for the objects which the player is carrying, room 1. That's five rooms altogether, which we shall number in this fairly obvious fashion (although the numbering is totally arbitrary):



First on paper and then in the program we must list which rooms have exits and where they lead, and allocate a message and message number to each room.

Room 1 (the player's carried objects) has no exit, of course, but it does have a message associated with it. In order to describe what a player is carrying, we must be able to say 'You are carrying:'. . So let this be the message for room 1. So, on one piece of paper we can create a pencil version of the room database, as follows:

Room	North	East	South	West	Message
1	0	0	0	0	1
2	5	0	0	3	2
3	4	2	0	0	3
4	0	0	3	0	4
5	0	0	2	0	5

As in Part 2, we have used 0 for an exit which doesn't exist, and the other numbers to indicate an exit's destination room. We must also write relevant messages (on another sheet of paper):

1 "You are holding:"
2 "You are in a crumbling room full of cobwebs. A passage leads west."
3 "You are in a cheerful pink room. The word BLAH is inscribed on the ceiling. A passage leads east, and a doorway full of whirling knives leads north."
4 "You're to the north of the pink room in an alcove. The only exit seems to be back the way you came."
5 "You're in a vast treasure chamber, with an exit south."

These messages do not exhaust everything we know about the rooms. What about that barred door? And what if the player isn't carrying anything? What can come after 'You are holding:'? Clearly we need messages 6 and 7:

6 "A barred door bars a north exit."
7 "Nothing"

Although all these messages will eventually become part of the program, for two reasons I do not recommend writing them directly into the program. First, let the program worry about where to find and store the messages. Second, you may well need to change the messages (i.e. find a mistake) as you go

along. Pencilled writing is a lot easier to modify than program structure.

However, since the static database is unlikely to be modified, we can define that part which relates to our rooms. We'll put it in a DATA statement for now, at line 4000:

```
3990REM room data
4000DATA 0,0,0,0,1
4010DATA 5,0,0,3,2
4020DATA 4,2,0,0,3
4030DATA 0,0,3,0,4
4040DATA 0,0,2,0,5
```

Each line corresponds to a room: 4000 to room 1 (the player); 4010 to room 2 (cobweb); 4020 to the pink room; 4030 to the alcove; and 4040 to the treasure vault. The first four numbers in the data statement are the rooms reached by going North, East, South and West respectively from the room corresponding to that data line. The snags (i.e. the exit programs) haven't shown up yet. The last number on each line tells the program which message goes with that room. You might think, by the way, that as these run 1 to 5, there is no need to store these numbers. But on a more involved scenario, the numbering almost certainly would get more complicated – so the message numbers are needed.

We must now set up a data list to hold the messages relating to objects:

```
4490REM object data
4500DATA 8
4510DATA 9
4520DATA 10
```

which relate, sequentially, to objects 1, 2, and 3. These three messages are:

8 "A shield"
9 "A black rod"
10 "A jewelled crown!"

(by convention, 'treasures' get exclamation marks; sometimes this makes them easier to spot). We shall

need another message (11) to give an extra line after the shield description when the shield is being worn:

11 "(which you are wearing)"

So when the shield is merely carried, message 8 occurs. When the shield is worn, message 8 followed by message 11 will occur. The dynamic part of the object database is so short that it can be done explicitly at the beginning of the program, so let's leave that for a minute.

The next bit of pencil-and-paper work concerns the vocabulary. You may feel impatient to start programming, but since most of the programming involves manipulating the vocabulary, we really do need that defined first. Start by writing a list of verbs, together with their two key numbers (the command number, and the command type) in alphabetical order. This bit is important. We shall see that vocabulary searches are much faster when the data is arranged alphabetically. I shall now list all the verbs I intend the program to understand. It is instructive to make your own list first, and compare it with mine to see how much disagreement there is. Here's my list.

```
BLAH,1,0
DROP,2,1
E,3,0
EAST,3,0
GET,5,1
GO,4,2
INV,6,0
LOOK,7,0
MOVE,4,2
N,8,0
NORT,8,0
Q,9,0
QUIT,9,0
S,10,0
SAY,11,2
SOUT,10,0
STOP,9,0
TAKE,5,1
THRO,2,1
W,12,0
WAVE,13,1
WEAR,14,1
WEST,12,0
```

Twenty-three verbs in all, each truncated to four letters. You may not have thought of including synonyms, but they are important. Hence 'E' and 'EAST' both appear; so do 'GO' and 'MOVE', and 'Q', 'QUIT', and 'STOP'.

Let's examine the list. The first number of each command arose according to the position it originally occupied as I wrote down the list; there's no particular necessity to do this. (Indeed, GO and GET occurred out of alphabetical order when I first wrote it, so their numbering is, in that sense, out of sequence).

Finally, as already explained, each verb has its type after its number: so 'BLAH' is of type 0 – no second word – and so on.

Putting this part into the static database is trivial. We'll use line 3000 for this.

Synonyms are easy to spot by their carrying the same number as each other. 'DROP' and 'THRO(W)' are synonyms for this game and thus both are numbered 2. In all there are 14 different commands possible.

```
3000DATA BLAH,1,0,DROP,2,1,E,3,0,
EAST,3,0,GET,5,1,GO,4,2,INV,6,0,
LOOK,7,0,MOVE,4,2,N,8,0,NORT,8,0,Q,
9,0,QUIT,9,0,S,10,0,SAY,11,2,SOUT,
10,0,STOP,9,0,TAKE,5,1,THRO,2,1,W,
12,0,WAVE,13,1,WEAR,14,1,WEST,12,0
```

When we initialise the program, this is ready to be read into an array. Do not just leave it as data.

Now come the things:

```
CROW,3
JEWE,3
ROD,2
SHIE,1
```

Again these are listed in alphabetical order. Notice the double occurrence of words meaning crown. These also go into a data statement, this time at line 3100:

```
3100DATA CROW,3,JEWE,3,ROD,2,SHIE,1
```

We follow this with our list of special words:

```
DOOR,1
KNIV,2
```

which fit into line 3200:

```
3200DATA DOOR,1,KNIV,2
```

3.8 Writing the program (2) – the main program

The next part of the program to write is the main loop, as described in Section 3.6. First, let's deal with initialisation:


```
10DIM rs%(5),os%(3),or%(3),v$(23),v%(23),vt%(23),o$(4),o%(4),s$(2),
s%(2)
20MODE7:nr=5:no=3:nv=23:nt=4:ns=2:x$=STRING$(20," "):y$=x$
30FOR I%=6 TO 7:PRINT TAB(9,I%);CHR$141;"Mini-adventure":NEXT
40PRINT'''':R%=2:O%=0:W%=12:or%(1)=2:or%(2)=4:or%(3)=5
50FOR I%=1 TO nv:READ v$(I%),v%(I%),vt%(I%):NEXT
60FOR I%=1 TO nt:READ o$(I%),o%(I%):NEXT
70FOR I%=1 TO ns:READ s$(I%),s%(I%):NEXT
```

Line 10 sets us some arrays to hold both the dynamic part of the database and also some bits of the static database (the rest of which, remember, sits in DATA statements). The reason all the static database is not DATA'ed is for speed of command handling, as we'll see below.

The arrays are as follows:

```
rs% - room state (note initials)
os% - object state
or% - object's room
v$  - a string holding each of the verbs which is understood
v%  - a string holding which verb v$ is
vt% - what type of verb v$ is (i.e. any second words, etc.)
o$  - a string holding each of the things understood
o%  - which object a thing is
s$  - as o$, but for special words
s%  - as o%, but for special words
```

Line 20 sets mode 7 (never forget to do that!) and sets nr, no, nv, nt, and ns to the number of rooms, objects, verbs, things, and special words respectively. Get into the habit of always working with variables, as it's easier to modify the program when necessary. Line 20 also initialises the strings x$ and y$, which will eventually hold verb and object/special input. The idea of initialising them to rather long strings is to save space! Originally, BASIC will decide on a default length for any string if you don't give it one. If at some later time, the string has to stretch beyond that length, BASIC will throw away that bit of storage area, as the string doesn't now fit, and make some more elsewhere. If x$, say, increases in length several times during the game, this can waste quite a bit of room. So, by setting the strings long at the start, we guarantee that their storage area will not be reduced later.

Line 30 merely gives us a double-height title (ignore these characters on the Electron). Line 40 initialises some more variables:

```
R%  - the current room of the player (originally 2)
Q%  - the previous room of the player (set to zero!)
W%  - a number to count which of the 'water rising' messages is
      printed each turn. More on that below
or%(1 and 2 and 3) - set to the rooms each object is in (2,4,5).
```

The occurrence of that Q% may be puzzling, but in order to know whether he has moved, we need to compare R% (where he is now) with where he was before. This is stored in Q%, whose initial value doesn't matter provided it isn't 2.

Lines 50 to 70 read in the verb, thing, and special information from the data statements which we provided previously at lines 3000 to 3200.

After initialising, we proceed to the 'between turns' portion of the main program:

```
100REM between turns
110REPEAT IF Q%<>R% PROCdescroom(R%)
120Q%=R%
```

Line 110 begins the main game loop with a REPEAT, and then checks to see if the player has moved; if so, his new room is described. Then Q% is updated to R% whether or not he's moved. This has introduced a new procedure to describe the player's room, and we'll write that later. Notice that R% is a parameter for the procedure, as we might want to describe another room, for some reason.

Now we move to listening to the player:

```
200F%=0
210REPEAT
220IF F%=0 REPEAT INPUT'":"y$:UNTIL y$ <> ""
230J%=INSTR(y$," "):IF J%=0 x$=LEFT$(y$,4):y$="":GOTO 300
240x$=LEFT$(LEFT$(y$,J%-1),4):y$=RIGHT$(y$,LENy$-J%)
250IF LEFT$(y$,1)="  " REPEAT y$=RIGHT$(y$,LENy$-1):UNTIL LEFT$(y$,1)<>
" "
260y$=LEFT$(y$,4)
```

Here's the logic. First clear a flag variable, F%. This will sometimes be changed by the command subprograms below so as to indicate some special action: to 9 for a fatal act, to 2 if a new game is needed, and to 1 if the first word should be ignored and the second treated as a first word (e.g. GO WEST). By the way, we can't handle this last case in lines 200 to 400 as

other actions may be needed (GO needs a 'go where?' request before reprocessing can continue, for example). Line 210 sets up another REPEAT loop, to be terminated when sensible input has been acquired. Line 220 prompts the player with a colon provided F% is still zero (the only time this won't be so is if the first word has been disregarded, and a new vocabulary check being done). Line 220 gets a line of input (y$), with leading blanks omitted courtesy of BASIC. If the player didn't say anything, ask again. Line 230 looks to see if a space was part of y$ (notice that we can't be afflicted by the INSTR bug in BASIC I because y$ must be at least one character long). If there is no space in y$, the player typed a single word only. So we catch the first 4 characters of it and deposit them in x$,put nothing in y$, and skip on to the command decoding. Assuming there was more than one word (line 240), we first split before the blank and then take the first 4 characters and put them in x$ again. (Why two sets of LEFT$? Ask yourself what happens if the player types 'GO N'!) The remainder of y$ is left in y$.

Unfortunately, we don't know how many blanks the player typed between verb and second word, and BASIC's input didn't help there. So line 250 is devoted to user-friendliness. We shall not penalise the player for typing 'GET ROD' instead of 'GET ROD': this line pulls off all leading spaces. We don't have a REPEAT WHILE facility, but can get round it with an IF as shown. Finally, line 260 gets the first 4 characters and leaves them in y$.

At the end of all this, we arrive at command checking with x$ holding the first four letters of the verb typed, and y$ holding the first four letters of the second word, if there was one (otherwise it's empty). The command checking now proceeds (with the help of some more procedures):

```
300REM command checking
310PROCc:IF C%=0 PRINT "Eh??":UNTIL FALSE
320IF D%=0 AND y$<>"" PROCm(12):UNTIL FALSE
330IF y$="" O%=0:S%=0 ELSE PROCo:PROCs:IF D%=1 AND O%+S%=0
PROCm(12):UNTIL FALSE
350 UNTIL TRUE
```

Line 310 calls PROCc (c for 'check') which examines x$ and sees if it's on the list v$ of verbs. It returns two numbers: C% (the number of the verb) and D% (its type). If it couldn't find the verb, C% is zero. We'll have to write that later, too. So if the verb wasn't

understood, tell the player, and end the REPEAT loop started at 210.

By line 320, then, x$ is recognised. Now, does what was typed fit with the type of verb? Line 320 itself looks to see if there was a second word when none was allowed (verb of type zero). If so, print message 12 and back for another try. We rapidly jot down message 12 on our list:

12 "I don't understand that!"

(Why both 'Eh??' and 'I don't understand that!'? I use 'Eh??' to indicate that the verb wasn't understood, and 'I don't..' when the second word either wasn't understood or wasn't allowed. The player will be told this in the rules, and if it makes his life easier that's all the better – user-friendliness again.)

Line 330, if there was no second word, sets O% and S% (the values of object and/or special word respectively) to zero, and continues. The more detailed checking, in this case, will be done by each command subprogram.

If there was a second word, PROCo and PROCs are now called, which, rather like PROCc, set O% or S% respectively to the value of the object or special word represented by the second word. If the verb is of type 1 (must have a recognised second word), and it wasn't recognised, give out message 12 and try again.

Now that the command is understood, we can acknowledge it with an UNTIL TRUE to finish the REPEAT loop, and process it in lines 500 to 599:

```
500REM commands
510F%=0
520ON C% GOSUB ... somewhere!
530IF F%=9 PROCdie
540IF F%=1 THEN 210
550IF F%=2 PROCnewgame
```

This should be straightforward. Line 510 resets the marker flag to zero, because we'll be checking on its value in just a moment. Line 520, which cannot be finished yet, does a large GOSUB to the appropriate command subprogram. When we've programmed that, we can fill in the labels accordingly. Lines 530 to 550 then scan the marker flag, as discussed earlier. If the action was fatal (F%=9), off to the 'dieprog'. If

the first word should be ignored (F%=1) then jump back to 210.

(This finally reveals why the word is input originally to y$: it is so that the second word is still available for reprocessing as a first word. We cannot jump into a REPEAT loop, so we are forced to jump back to the start of it. But then, we don't want any input, hence the IF check on 220. This also explains why we had to reset F% at 510: 'GO' in a command like 'GO WEST' would have set F% to 1 on the first pass through, and directed the program to 210; but F% would still have been 1 at 500. So, we reset it before handling the commands.) Finally, if the player is trying to quit (F%=2), let him do so.

The post-program is equally simple, for this game:

```
600REM post-program
610IF or%(3)=1 PROCm(40):END
620W%=W%+1:PROCm(W%):IF W%=23 PROCdie ELSE UNTIL FALSE
```

Line 610 asks if the crown (object 3) is held by the player. If so, congratulate the player and end the game. This needs a new message, number 40. ('Why 40, and not 13?', you may ask. Well, we all make mistakes, and when I wrote this game I forgot to include the check on winning until after I'd written the other messages.) So, message 40 reads:

40 "The crown is yours! Well done!"

Line 620 adds one to the "water-marker" W%, and prints the appropriate message. If W% has reached 23, the player has run out of time and dies; otherwise continue around for the between-turns processing. We thus need a collection of messages numbered 13 to 23, of increasing urgency:

13 "There are faint sounds of water."
14 "The sounds are slightly nearer."
15 "The water sounds are quite loud."
16 "That water's getting nearer!"
17 "Water is soaking up through the floor!"
18 "Pools of water are on the ground."
19 "The water is ankle-deep!"
20 "The water is knee-deep!"
21 "The water is waist-deep!"
22 "The water is up to your chest."
23 "The water closes over your head. . ."

Notice how the last message sounds fatal – it should do, as it'll be followed by a word from the death and restart routine at line 1000:

```
1000DEFPROCdie:PROCm(24):PROCnewgame
1010DEFPROCnewgame:PROCm(25):INPUT x$:IF LEFT$(x$,1)<>"N" THEN RUN ELSE END
```

Line 1000 delivers a suitable death message:

24 "You have departed this world, alas."

and then continues to the restart routine (cunningly combining the two into one piece of program) at 1010. This asks about a new game, and checks the player input to see whether he said no. The restart is affected by RUN, otherwise the game ends. Both ends of PROCnewgame leave REPEAT loops unfinished, but RUN and END both handle those very efficiently! We write message 25 to complete the running program's messages:

25 "Would you like another game?"

3.9 Writing the program (3) – the command subprograms

A program of this level of simplicity enables us practically to write the command subprograms as we go along. Unless there are convenience reasons why we shouldn't, we might as well write and present them in the order we used when alphabetising. We start with the magic word, BLAH:

```
2000REM blah
2010IF R%<>4 PROCm(26) ELSE PROCm(27):R%=2
2020RETURN
```

Whenever the player is in room 4, BLAH will take him and what he is carrying to room 2; otherwise nothing will happen when he uses the word. Line 2010 expresses this. Here are the messages:

26 "Nothing happens"
27 "There is a fanfare of cream horns, and you are thrown through the air!"

The next program is DROP or THROW:

```
2030REM drop, throw
2040IF O%=0 PROCm(28):RETURN
2050IF or%(O%)<>1 PROCm(29) ELSE PROCm(30):or%(O%)=R%:os%(O%)=0
2060RETURN
```

Line 2040 checks if an object was the second word. If not, we mutter at the player:

28 "You can't do that!"

and return. In 2050 we think negatively once more. Is the player not holding the object? If so, say so (message 29):

29 "You're not holding that!"

and return. (Notice the vague 'that' in message 29; it handles singular and plural objects without caring. 'It' wouldn't work if the object were a bunch of keys, for example.) Otherwise, acknowledge with an 'OK', and move object O% to the player's room (R%). Reset the object's state to zero. (Why? Because it might be the shield, now not being worn.) Then return. It's hardly worth it, but here's message 30:

30: "OK"

Next comes EAST:

```
2070REM east
2080PROCroomdata(R%)
2090IF e%=0 PROCm(32) ELSE R%=e%
2100RETURN
```

Line 2080 calls an unwritten procedure PROCroomdata(R%). This hands back the destinations of the room's exits in n%, e%, s% and w%, together with the appropriate message for the room in m%. (This is clearly an all-purpose procedure.) Line 2090 looks for an exit. If there isn't one, say so (message 32) and return, else move the player there. Notice that there aren't any exit programs on east-bound traffic in this game. Message 32, another all-purpose one, reads:

32: "You can't go in that direction!"

(Message 31 I wrote at another time; it'll turn up.) The awkward word GO comes next:

```
2110REM go
2120IF y$="" PRINTx$;" where?":INPUT y$
2130F%=1:RETURN
```

but surprisingly it's easy to handle. If there is no second word (2120) ask for one in y$; if there is one, it's in y$ already. Set the F% flag (2130) and return. I hope you can now appreciate the uses of flags.

GET and TAKE come next:

```
2140REM get, take
2150IF O%=0 PROCm(28):RETURN
2160I%=or%(O%):IF I%=1 PROCm(33):RETURN
2170IF I%<>R% PROCm(34):RETURN
2180or%(O%)=1:PROCm(30):RETURN
```

Again, if there is no object, 2150 uses the same message and returns. Line 2160 loads I% with O%'s room, as we'll be using it several times. Negative thinking now applies. If the room is 1, the player's already holding O%, so say so (message 33) and return. (Don't forget all these returns; you can get some funny messages otherwise.) If the object isn't in room R%, say so as well, (message 34) and return. Otherwise (2180) put it in room 1, acknowledge with 'OK', and return. Thus we need messages:

33 "You're already holding that!"
34 "That's not here!"

both of which use the neutral 'that' again. INV (inventory) and LOOK (repeat room description) are trivial (we merely pass the buck to later on):

```
2190REM inventory
2200PROCdescroom(1):RETURN
2210REM look
2220PROCdescroom(R%):RETURN
```

by simply calling PROCdescroom (already mentioned). NORTH is more awkward, because of the exit programs:

```
2230REM north
2240PROCroomdata(R%)
2250IF n%=0 PROCm(32):RETURN
2260IF R%=2 AND rs%(2)=0 PROCm(6):RETURN
2270IF R%<>3 R%=n%:RETURN
2280PROCm(35):IF or%(1)<>1 PROCm(36):F%=9:RETURN
2290IF os%(1)=0 PROCm(37):F%=9:RETURN
2300PROCm(38):or%(1)=0
2310R%=n%:RETURN
```

First, in 2240 we pick up the room data. If there's no exit (2250), we say so and return. Now the player can exit, what can go wrong? If the room is 2 (2260) and the rod hasn't been waved, the door is still there. So we say so with message 6 (carefully written to serve the double purpose of description and a blocking message), and return. Unless the player is now in room 3 (2270) we may move him to n%, and return. So what of the knives? Line 2280 tells of the knives, whatever the circumstances:

35: "The knives stab at you as you pass."

This does not say what happens to the player. . .So, if he wasn't carrying the shield (2280 continued) we kill him (F%=9), say so (message 36), and return:

36 "They slice you to ribbons."

By 2290, the player must at least be carrying the shield. If he wasn't wearing it (2290) we also kill him, but give him the hint message 37:

37 "You're not actually wearing the shield, so the knives get you."

If he survives all that, he deserves to get through! So we congratulate him at line 2300 with message 38:

38 "They bounce off the shield, which shatters."

(notice how each of the messages 36 to 38 follows on after 35). Then we have to get rid of the shield.

Where can we put it? On the face of it, there is no appropriate room. Fortunately, BASIC's dimensioning handles it for us, because the index for an array starts at zero, not one. Hence room zero exists; and in it we place the shield!

See how easy it is to fail to set up a database flexible enough for the job at hand, and how important it is to keep one extra room (zero) to hold all objects which must be disposed of for one reason or another.

Back to the player. When we left him (line 2300) he still hadn't gone through the knives. So we destroy the shield, and then move him at 2310.

The next on the list is QUIT:

```
2320REM quit, stop
2330F%=2:RETURN
```

Here we set F% accordingly, and return. After that comes SOUTH, mercifully simpler than NORTH:

```
2340REM south
2350PROCroomdata(R%)
2360IF s%=0 PROCm(32):RETURN
2370IF R%=4 PROCm(35):PROCm(36):F%=9:RETURN
2380R%=s%:RETURN
```

Again we get the data for R% (2350) and check for no exit (2360). The only room giving problems is 4 (the

knives again). We mention them (2370) with message 35 (doing yeoman service!) and slice the player with message 36 (he can't have the shield because he lost it getting in). The F% flag is set, and that's the end of the player. Otherwise, we just move the player at 2380. You may be wondering why we don't give the player a message when he moves. That's because he automatically gets a new room description during the between-moves procedure.

SAY is a strange command:

```
2390REM say
2400IF y$<>"BLAH" PRINT "OK, '";y$;"'!":RETURN
2410GOTO 2000
```

If the second word wasn't BLAH – which has its own command sequence – then we print an odd message which substitutes the word the player typed (y$). So if the command was SAY HELLO, out comes "OK, 'HELLO'!" If the second word was BLAH, though, we jump (not GOSUB) to the BLAH routine at 2000. Then the RETURN at the end of the BLAH routine takes us back to the main program. Although GOTOs are messy, this one avoids repeating program modules.

The last of the directions is WEST:

```
2420REM west
2430PROCroomdata(R%)
2440IF w%=0 PROCm(32) ELSE R%=w%
2450RETURN
```

which presents no problems as, like EAST, there aren't any exit programs to worry about. Next comes WAVE:

```
2460REM wave
2470IF O%=0 PROCm(28):RETURN
2480IF or%(O%)<>1 PROCm(29):RETURN
2490IF R%<>2 OR rs%(2)>0 OR O%<>2 PROCm(26):RETURN
2500PROCm(39):rs%(2)=1:RETURN
```

At line 2470 we check if an object was mentioned; at 2480 whether the object is held by the player. Line 2490 now checks negatively for any reason waving an object might produce no result: wrong room (R%<>2), wrong state of room 2 (rs%(2)>0, i.e. already waved) or wrong object (O%<>2). In any of these cases, use the 'nothing happens' message. Otherwise (line 2500) remove the door (rs%(2)=1) and say so with message 39:

39 "The door vanishes quietly."

The last command is WEAR:

```
2510REM wear
2520IF O%=0 PROCm(28):RETURN
2530IF or%(O%)<>1 PROCm(29):RETURN
2540IF O%<>1 PROCm(28):RETURN
2550os%(1)=1:PROCm(30):RETURN
```

which is simple enough. Is it an object (line 2520)? Is the object held (line 2530)? Is it the shield (line 2540)? Let the player wear it (state goes to 1) in line 2550.

This concludes the commands section of the program. Our last chore here is to return to line 520 and fill in that list of GOSUBs:

```
 5200N C% GOSUB 2000,2030,2070,2110,2140,2190,2210,2230,2320,2340,2390,
2420,2460,2510
```

3.10 Writing the program (4) – the utility procedures

All that remains is to write the various procedures referred to in the main playing program. The first is the ubiquitous PROCdescroom. Here it is:

```
4990REM helpful procedures
5000DEFPROCdescroom(R%)
5010PROCroomdata(R%):PROCm(m%):IF R%=2 AND rs%(2)=0 PROCm(6)
5020m%=TRUE:IF R%<>1 PROCm(31)
5030FOR I%=1 TO no
5040IF or%(I%)=R% PROCdescobj(I%):m%=FALSE
5050NEXT
5060IF m% PROCm(7)
5070ENDPROC
```

Line 5010 obtains the room data for R%; we only need the message number m% here. This is printed out, followed by message 6 (the door message) in the right room and under the right circumstances. (Do you find this checking rather ugly? I do hope so, as we'll improve on this later.) We now use m% temporarily as a logical flag, as it's served its main purpose. Line 5020 sets it to TRUE and, unless we're describing the player, prints out the last of our messages, number 31:

31: "You can see:"

(whose number shows that I didn't write the game in the order I'm describing it. The point is, it doesn't matter what order you write it in as long as it's logical to you.)

The point of message 31 is that we have to tell the player about the objects in view. This technique is, I think, clumsy; it, too, will be improved later. Lines 5030 to 5050 run through the objects. If any are in R%,

we describe them (with PROCdescobj, whose function should be obvious), and set m% to false. This last will show that at least one object was mentioned. Thus, at 5060, if m% is still TRUE, we print 'Nothing' just to finish the sentence off.

This procedure, and others, mentioned PROCroomdata. This is easy:

```
5100DEFPROCroomdata(R%)
5110RESTORE (4000+10*(R%-1)):READ n%,e%,s%,w%,m%
5120ENDPROC
```

Line 5110 restores data reading to the appropriate line (we were working in multiples of 10 for room data, remember), and reads in the four destinations and the message number.

PROCdescobj was used above:

```
5200DEFPROCdescobj(O%)
5210PROCobjdata(O%)
5220PROCm(m%):IF O%=1 AND os%(1)=1 PROCm(11)
5230ENDPROC
```

This takes as argument the object number being described. It obtains the object data in 5210, reading the object message into m%. This is printed in 5220, along with the 'which you are wearing' message for the shield in the right circumstances. This, too, is ugly programming; more on it later.

This, in turn, referred to PROCobjdata:

```
5300DEFPROCobjdata(O%)
5310RESTORE (4500+10*(O%-1)):READ m%
5320ENDPROC
```

which again finds the appropriate data line and gets the message number. Messages themselves are handled by PROCm:

```
5330DEFPROCm(m%)
5340RESTORE (4800+m%):READ a$:PRINT a$:ENDPROC
```

which picks up the appropriate string in a$ and prints it.

Now only three procedures are left, each one referred to only in lines 300 to 400. These are PROCc, PROCo, and PROCs. Each procedure takes either x$ (for PROCc) or y$ (for the other two) and attempts to match it to one of the pieces of vocabulary. When programming the main loop, we should have written

down just what these procedures were supposed to pass back, in the event either of failure or of success. You probably can't remember now; hence the necessity for jotting things down in an orderly fashion.

To remind you, PROCc hands back C% as the verb number (or zero in the case of failure) and D% as the verb type. PROCo and PROCs hand back O% and S% respectively as the object or special word, with zero for failure. All three procedures are alike, so only PROCc will be described in detail.

The procedure uses the fact that the verb data is alphabetical to speed up the search for a match. The method, or algorithm, used is what's called a 'binary search'. It's like those guessing games when one player thinks of a number between 1 and 100 and the other player must guess it. You try 'Is it less than 50?' first; either answer reduces the area of search to half what it was before, unless the guess was spot on. One continues halving the area until the guess is correct; or, in our case, if it has failed.

The point of doing such a search is speed. With 23 verbs to check, just going through and checking one by one until you succeeded or passed the command typed would need on average 12 checks. This procedure makes that only 7. Not much difference, you may think. But on 63 verbs, simple checking takes 32 checks on average; binary searching takes only 8 at most.

```
5350DEFPROCc
5360C%=0:H%=nv:U%=1
5370IF x$<v$(U%) OR x$>v$(H%) ENDPROC
5380IF x$=v$(U%) C%=U%:GOTO 5440
5390IF x$=v$(H%) C%=H%:GOTO 5440
5400IF H%-U%=1 C%=0:ENDPROC
5410C%=(H%+U%) DIV 2:IF x$=v$(C%) THEN 5440
5420IF x$<v$(C%) H%=C%:GOTO 5400
5430U%=C%:GOTO 5400
5440D%=vt%(C%):C%=v%(C%):ENDPROC
```

Line 5360 sets C% to zero in case we don't find anything, and H%, U% to the highest verb number (nt) and lowest (1) respectively. As we zero in on the verb, these will change. Line 5370 checks to see if the verb typed is outside the range of H% and U%; if so, we return empty-handed. Lines 5380, 5390 actually check the first and last verbs for a match, setting C% accordingly in a procedure PROCsetc below.

We start the hunt with a repeat loop at 5400. Now, if H% and U% have become too close to each other, we missed the verb,end the loop and return. Initially, of course, this hasn't happened. At 5410, we try halfway between H% and U% (using C% for this value); if successful, we set C% and end. Otherwise (5420) we halve the area of search depending on whether x$ was less (alphabetically) than the current v$, or more than v$. In each case we reset one of H% or U% and continue the loop. Finally, the procedure at line 5440 sets the appropriate D%, then overwrites C% with its value from the table of verbs.

As I said, PROCo and PROCs, which check the vocabulary for objects and specials, are very similar. Here they are, without any further explanation:

```
5450DEFPROCo
54600%=0:H%=nt:U%=1
5470IF y$<o$(U%) OR y$>o$(H%) ENDPROC
5480IF y$=o$(U%) O%=U%:GOTO 5540
5490IF y$=o$(H%) O%=H%:GOTO 5540
5500IF H%-U%=1 O%=0:ENDPROC
55100%=(H%+U%) DIV 2:IF y$=o$(O%) THEN 5540
5520IF y$<o$(O%) H%=O%:GOTO 5500
5530U%=O%:GOTO 5500
55400%=o%(O%):ENDPROC
5550DEFPROCs
5560S%=0:H%=ns:U%=1
5570IF y$<s$(U%) OR y$>s$(H%) ENDPROC
5580IF y$=s$(U%) S%=U%:GOTO 5640
5590IF y$=s$(H%) S%=H%:GOTO 5640
5600IF H%-U%=1 S%=0:ENDPROC
5610S%=(H%+U%) DIV 2:IF y$=s$(S%) THEN 5640
5620IF y$<s$(S%) H%=S%:GOTO 5600
5630U%=S%:GOTO 5600
5640S%=s%(S%):ENDPROC
```

I also hope you find it ugly that we have written three almost identical procedures when (surely?) we could have made one do the work.

Well, that's the program written. All we have to do now is to transfer those messages off paper and into the program:

```
 4800REM messages
 4801DATA "You are holding:"
 4802DATA "You are in a crumbling room full of     cobwebs. A
passage leads west."
 4803DATA "You are in a cheerful pink room. The    word BLAH is inscribed
on the ceiling.  A passage leads east, and a doorway
full of whirling knives leads north."
 4804DATA "You're to the north of the pink room in an alcove. The
only exit seems to be    back the way you came."
 4805DATA "You're in a vast treasure chamber, with an exit south."
 4806DATA "A barred door bars a north exit."
```



```
 4807DATA "Nothing"
 4808DATA "A shield"
 4809DATA "A black rod"
 4810DATA "A jewelled crown!"
 4811DATA "(which you are wearing)"
 4812DATA "I don't understand that!"
 4813DATA "There are faint sounds of water."
 4814DATA "The sounds are slightly nearer."
 4815DATA "The water sounds are quite loud."
 4816DATA "That water's getting nearer!"
 4817DATA "Water is soaking up through the floor!"
 4818DATA "Pools of water are on the ground."
 4819DATA "The water is ankle-deep!"
 4820DATA "The water is knee-deep!"
 4821DATA "The water is waist-deep!"
 4822DATA "The water is up to your chest."
 4823DATA "The water closes over your head..."
 4824DATA "You have departed this world, alas."
 4825DATA "Would you like another game?"
 4826DATA "Nothing happens"
 4827DATA "There is a fanfare of cream horns, and  you are thrown
through the air!"
 4828DATA "You can't do that!"
 4829DATA "You're not holding that!"
 4830DATA "OK"
 4831DATA "You can see:"
 4832DATA "You can't go in that direction!"
 4833DATA "You're already holding that!"
 4834DATA "That's not here!"
 4835 DATA "The knives stab at you as you pass."
 4836DATA "They slice you to ribbons."
 4837DATA "You're not actually wearing the shield, so the knives get you.
 4838DATA "They bounce off your shield, which      shatters."
 4839DATA "The door vanishes quietly."
 4840DATA "The crown is yours! Well done!"
```

The spacing on some of these looks a little odd, but it's only to ensure that when that particular string is printed out, the lines don't do anything nasty when 40 characters are reached.

3.11 Afterthoughts: improvements and debugging

I hope several possible improvements occurred to you during that long description. There were several quite crude features. For example, what of that 'You can see:' message, followed by 'A shield'? Surely it would be better to store a rather more descriptive message about that shield, like 'There is a battered Saxon Ex-Army Surplus shield lying slumped on the floor'?

But there's a snag there. When it's being carried by the player, we don't want a message like:

You are carrying:
There is a battered Saxon Ex-Army . . .

which means that two different messages about the

shield are needed. The second one, for use when carried, would just be 'A shield', as we had it before.

Perhaps you'd have liked a score procedure (on a game that short?) Feel free to write one – award points for problems solved. You can count these easily by checking where the shield is, what state it's in, whether the door is there, and so on.

The real reason for this section, however, is to talk about debugging. If you've typed this program in, it's fairly likely that it won't work. If you've just programmed a simple game, it's unlikely to work first time either. Don't worry – this game didn't work the first (or even the twentieth) time for me. To begin with, I fell into the standard trap and used OR% instead of or%; BASIC took this as OR. This caused a lot of hunting through the program weeding out every occurrence of OR%! Then I had to check each message to see that its layout was acceptable. Fortunately, BBC BASIC is flexible, and a simple loop in immediate mode like:

```
f FOR I% = 1 TO 40: PROCm(I%) :X=GET:NEXT
```

sufficed for checking, independent of the rest of the program. (About half the messages needed adjusting, if you're interested.) Several other things went wrong, too; I discovered the lack of the way of ending, which necessitated message 40, and so on. I've left these in, rather than covering my tracks, because real-life programs are often messier than you or I would think. So don't be afraid if yours don't work first time; neither do mine!

You can do a lot of checking of the various procedures in immediate mode:

```
x$="ABCD":PROCc:PRINT C%
```

and so on. But much of the checking of a game like this must be done during play. A good trick, for example, is to add an extra line around line 100 which prints out the values of all the arrays, R%, etc. And if all else fails, there's always TRACE ON. Or print out a silly message at some point, to test if your program is getting there or not. There are many such methods, and all are worth trying if you get into difficulties. In the next game, I'll show you a very useful trick involving two commands the player doesn't know about – purely as debugging aids.

4 CREATING AN ADVANCED ADVENTURE GAME: 'ROMAN'

4.1 An overview of the rest of this book

Before we start to create an advanced Adventure, it's worth sketching out the slightly complicated path we have to take.

In this Part we shall plot the Adventure, and decide upon the final version of the database system. This is designed to take up as little space as possible to give maximum room for programming. In order to achieve this a separate program must be created to feed the data into the database. This program is created in Part 5.

Part 6 creates a 'shell' of the main program; this can be used for any future games you may create.

Part 7 fills in the shell for 'ROMAN', and shows how to organise all the material into a single game. You may find the appendices useful to consult on some awkward matters.

4.2 Plot development

Now we have the material at hand, we can begin to think about writing a fairly advanced game. As before, we'll begin with the plot and the puzzles involved. If you have sent for the cassette tape which goes with

this book (see page 1), you should play the game ('ROMAN') before continuing reading. I hope you enjoy it!

Adventure games often arise from concepts – more about this in Part 8. In this case, the concept was to replace the usual fantasy setting by historical reality, namely Caesar's Rome. Ancient Rome itself generated a great deal of the plot; some was deliberately designed to illustrate how to handle certain programming problems; and some simply emerged as the design progressed.

Because all Adventures need a point and this game has a historical setting, I cast around for a suitable quest for the player. Acquiring treasure is fun, but why would the average Roman go treasure hunting? Suppose he needed money? O.K., but in that case treasure-hunting might seem extravagantly adventurous. It was at this point that the idea of owing money to some scheming Roman senator came to mind. The player will be bound to pay back the debt, and that will justify involving him in a quest for treasure. So there is our motivation. In addition, we can almost certainly arrange a puzzle or two based on the actual paying back of the debt.

What puzzles, appropriate to our setting, come to mind? One immediate source is the use of ancient words. Suppose we chose objects whose names are not immediately recognisable to modern readers? A pilum, for example, was a long military spear. If we refer to it as a pilum, the player will probably not know its use, which is all to the good (N.B. we must play fair, however, and also allow 'spear' as an acceptable piece of vocabulary). Again, a gladius was a Roman short sword. . .two weapon names suggests some killing, don't they? If the player has looked up gladius and pilum, he should be thinking along the same lines. So we must remember to include a few objects or people which must definitely not be killed, as well as some that can.

What places come to mind when thinking about Rome? The Coliseum, the temples, and the Senate occurred to me. No real hope of much treasure lying about in the Coliseum, which was used for games, but let's come back to that later. The Senate is the obvious place in which we'd meet the senator, so no treasure

there either. That leaves the temple. Ah yes, how about a statue or bust? This gives us one treasure. How can we make getting the bust hard for the player?

Temples are mainly occupied by priests, who won't take kindly to players barging in and stealing busts. Somehow, priests don't sound safe to kill, either! So how could the player become 'acceptable' to a priest? If he were another priest perhaps. . .? That will presumably involve imitating a priest, either in dress or by action. Not being quite sure how priests dress, imitating an action sounds a good idea. How about a sacrifice of some animal? (N.B. We must provide some hints for the player, who has been given no reason to think along these same lines.) So, the player must bring with him a suitable animal for sacrifice; a chicken is the obvious candidate.

A sacrifice would also provide a use for the gladius (chopping off the poor bird's head). OK, we have a puzzle. Now how do we make it harder? Any or all of the following will do: (a) make getting the gladius more difficult; (b) make getting the chicken more difficult; or (c) make getting into the temple more difficult.

If in doubt, go for all three! Here's where we can introduce two different types of puzzle relating to obtaining objects: (1) leaving objects in places which are dangerous to get at (in this case, the gladius' location) or (2) making getting them dependent upon a whole chain of actions.

Let's put the gladius (the simpler to arrange) beyond a sheer drop. Jumping over a drop always worries players, and worry is a key ingredient of Adventure games. But to lull the player into a false feeling of confidence let's allow him to make the jump successfully. Only when jumping with the gladius (or pilum, come to that) will he miss his hold and fall into the abyss. The solution he must find is simple enough: first throw the gladius or pilum to the other side, then jump to safety.

A good Adventure game contains a mixture of easy and difficult puzzles. In turn, let's make getting the chicken hard! Obviously one can't just pick it up; anyone who has tried knows how hard one is to catch. We must force the chicken into the player's hands.

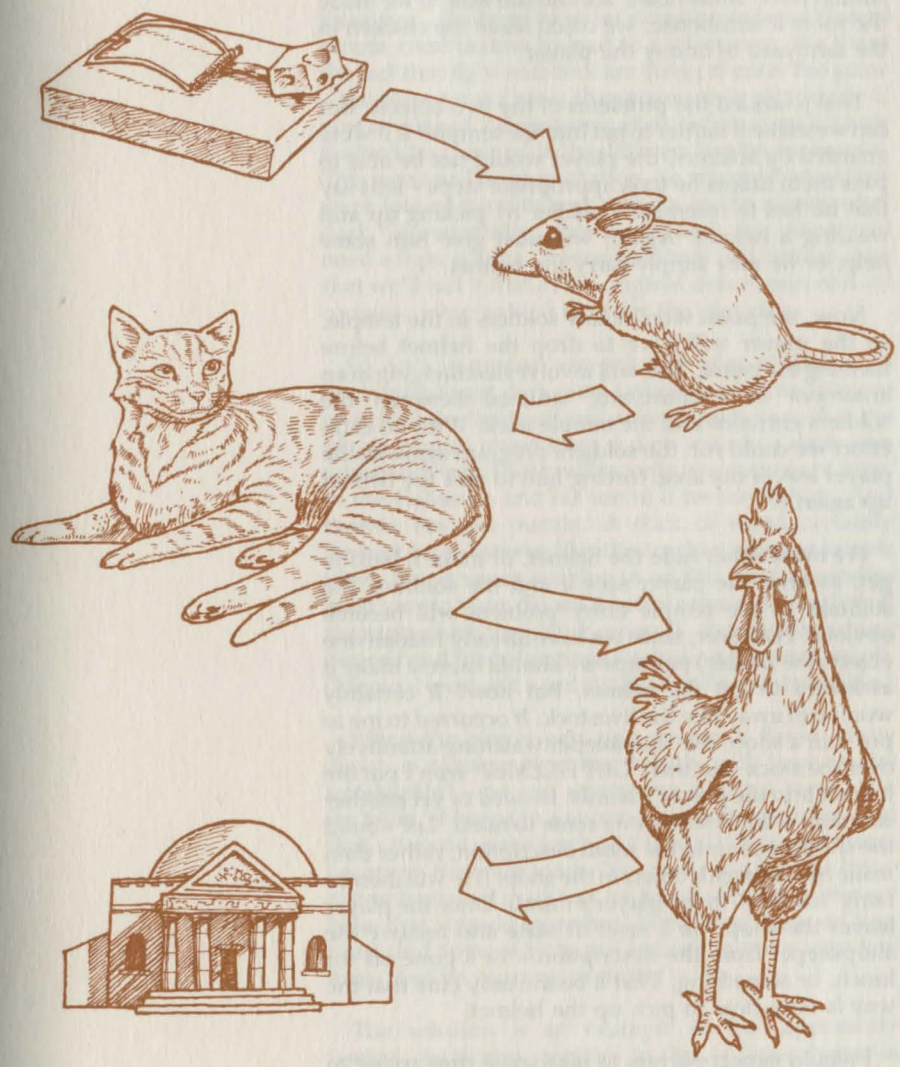
Throwing or dropping a cat will do nicely. When the chicken jumps with fright it can be caught.

But if we stop here, life will be too simple. Now, we must make the cat hard to get. Well, the same goes for picking up cats as for picking up chickens: if they don't want to be picked up, it can't be done. So, we need a new puzzle for cat-catching; of course we neither wish to repeat puzzles nor to bore the player. In any case cats don't scare easily. Perhaps we should attract the cat? A mouse could do that, but let's make it a dead one, else we'll have the problem of picking up the mouse as well, which may be taking a good idea a little too far. If the player is holding the mouse when he says 'GET CAT', the cat will jump into his arms, and eat the mouse.

The cycle now continues: to get the chicken, we need the cat; to get the cat we need the mouse; but where shall we put the mouse? Cunning must prevail. Suppose we left the chicken and the cat in easy reach to worry the player, who can't pick them up, but then left the mouse just a bit further off? It's easy to see that that wouldn't do at all. It takes very little mental effort to connect a dead mouse with a cat – we did it in one sentence just now. Once the player sees the mouse, the whole chain will become obvious. Thus we must delay discovery of the mouse as long as possible. From this we can derive a useful general rule: if you don't want a player to find something, don't leave it anywhere! Instead, make the player do something that will produce the mouse out of nowhere.

What action could produce a dead mouse? Laying a mousetrap comes immediately to mind. I presume the Romans had mousetraps; my informants can't tell me. We could disguise it by calling it a contraption of iron and wood – but in all fairness we should mention cheese as well, and let it also answer to 'TRAP' (not 'MOUSETRAP', which is indistinguishable from 'MOUSE' – it's advisable to note down petty details like that as one goes along). The action, then, is laying the trap and leaving it down; the player cannot simply drop the trap. He must leave the room and allow the mice to come and play. On his return we can present him with a dead mouse.

I hope this part of the scenario is taking shape in your mind. Thinking backwards is a standard part of



Adventure plot creation. The correct sequence is: find trap and leave somewhere. Return to collect mouse. Take to cat and get cat. Take cat to chicken, drop cat and catch chicken. Take chicken to temple, and, when confronted by priest, kill the chicken. The priest will leave, allowing the player to loot the temple.

A minor point – the place for the trap to be left should have 'small holes' around the wall. If we made the room a farmhouse, we could leave the chicken in the farmyard to annoy the player.

That's tackled the problems of the two objects. But can we make it harder to get into the temple? If it were guarded by soldiers, the player would not be able to pass them unless he took appropriate steps – let's say that he has to resemble a soldier by picking up and wearing a helmet. Again, we must give him some help, or he may simply carry the helmet.

Now, the priest won't allow soldiers in the temple, so the player will have to drop the helmet before meeting the priest. This will involve his changing in an anteroom or a courtyard, situated between the soldiers entrance and the temple itself. With no extra effort we could run the soldiers program again as the player leaves the area, forcing him to pick the helmet up again.

We must either hide the helmet, or make it hard to get, as once the player sees it and the soldiers, the solution to the temple entry problem will become obvious. However, since we have already hidden one object (the mouse) perhaps we should merely make it awkward to get the helmet. But how? It certainly won't run away like the livestock. It occurred to me to put it in a shop, the shopkeeper watching attentively over the stock. But then 'GET HELMET' won't put the helmet into the player's hands. In need of yet another solution, a fairly subtle one came to mind. Let's build the shopkeeper into the room description, rather than make him a specific object in the shop. (He will then be fairly 'low-key' in the player's mind). Once the player leaves the shop, we'll reset its state and remove the shopkeeper from the description – he's gone off for lunch, or something. That'll be the only clue that the way is now clear to pick up the helmet!

I would expect players to take some time trying to get the helmet; fail miserably; go off and do other things; come back later and pick it up with no difficulty; wonder why; and then, when trying the game all the way through, find that once again they can't pick up the helmet because by restarting, they have restored the shopkeeper! A simple puzzle, but reasonably elegant.

This convoluted chain leads only to one of three treasures – the bust. Now let's create a rather different puzzle combination to lead to a silver ring. We'll use the fact that light and dark are going to enter the game – the player must meet the senator only after dark. If you've played Adventure games before, you'll know that striding around in the dark tends to be dangerous. This game will be no exception – in ancient Rome there were lots of bandits and robbers in the streets after dark. So eventually, after it gets dark, the player will need a light source. A wooden torch will do fine. Not that we'll call it that in the original description of it, of course – why make it easy for the player?

So let's suppose we tell the player that it is beginning to get dark, and a number of turns later we switch all the lights off and arrange matters so that the player cannot move from a dark area to a dark area without dying. There will be only two sources of light: in the Coliseum and his torch, if he can light it. And therein lies the puzzle. A stick of wood certainly doesn't burn forever like the torches you see in epic films. No, it needs soaking in oil first, just as candles need wax to burn. So we'd better organise a pool of oil through which the player can walk, thereby drenching himself and his belongings (including the torch) in oil. (Memo – we don't want the pool to be inexhaustible.)

When the player gets to a source of flame – how about a brazier of coals, which will have to be untakeable? – he can attempt to light his torch. If it isn't oily, it burns to a stump well-nigh instantly, and that's the end of the torch. If it is oily, it'll catch fire and become a first-rate torch. No, that's too simple. If the player is soaked in oil too, he'll probably catch fire too! Now that sounds more fun. The player, now on fire, must find some way to put himself out in a very few turns, before burning to death!

The solution is an example of the 'apparently useless, fatal area' puzzle. In this, a 'no-go' area is provided in which the player dies on entering. He can waste a lot of time trying out ways of entering, none of which will work because he has to have achieved something to 'earn' safe entry. In this case, he must be on fire! We'll create a damp, misty area, where the player is assassinated by a runaway slave, unless he enters while on fire. Then he will he be safe, because the mist will condense onto his body and put the fire

out. The slave, revealed, will run away, making the area, and beyond it, accessible to the player. There we can leave a silver ring, stolen by the slave, which (a) provides a treasure and (b) congratulates the player on solving the problem. A nice puzzle – can we frustrate the player further, when he solves it? Yes indeed. If the mist is so active as to put out the player's fire, it should do the same to his torch! So the poor player, staggering around and on fire, will try the mist, but to his disappointment the torch will go out permanently too! The solution is trivial – he must drop the torch before entering the mist!

There will be one further light-and-dark puzzle, and I'll return to that at the end of this section, when we discuss the end of the game.

The third, and last, sequence of puzzles relates to the Coliseum. I thought for some time about how the player could find a treasure in the Coliseum, and finally decided that he should win a gold wreath in the Games being held there (after dark, just to put some structure on the game). So let's create puzzles related to entry and exit from Coliseums, together with fighting in the arena.

Entry to the Games will need money, clearly. The sesterce was a monetary unit, another suitably ancient word. 'Coin' and 'Money' will obviously have to be included in the vocabulary as synonyms, however. We can leave the sesterce in the anteroom to the temple. (I prefer my games to 'use' rooms fairly thoroughly, rather than creating a new one for each puzzle, or setting up large maps with fewer puzzles, but it's only a matter of preference). 'PAY SESTERCE', or any of quite a few alternatives, will gain entry to the Coliseum – we could mention a pay booth to plant the idea in the player's mind.

Having transferred the player inside the Coliseum, it is time for a maze! We haven't had any mazes yet, unless you count the whole of 'CAVES'. At its simplest, a maze is merely a collection of rooms, all of which have the same description. It is customary for the player to map such mazes by dropping an object in each room, thus making each room description unique. Well, if you want to have mazes like that, go ahead. They're bafflers if you've never come across them before. But for this game, we'll design two

slightly different mazes, neither of which will be mappable by object-dropping.

Once inside the Coliseum, the player will find himself surrounded by crowds forming a four-room maze of identically-described, interconnecting rooms. Dropping an object loses it among the feet of the crowd, thus preventing maze-mapping by object-dropping. However, one of the exits in each room leads in a unique direction – i.e. only one room has a north exit, only one an east exit, and so on. So by the shape of the pattern of permissible exits, the player will be able eventually to deduce first which room he's in, and second how to get through the maze. Note that there are only four rooms, as I loathe having to map 37 rooms unless there's a particular point to the structure of the maze. This won't make the solution to the maze any easier, as we shall make it hard to find the crucial exit accidentally.

Leaving the fourth room correctly will put the player in the central arena, with a roaring lion bearing down on him and a crowd egging him on! This, of course, is where the pilum is useful. 'THROW PILUM' (or, curse it, 'KILL LION' if he holds the pilum) will get rid of the lion – any other action results in the player's demise. His reward will be a gold wreath from Caesar himself, plus a hint as to how to get out of the arena! There will be a small gate, which will apparently dump the player back in the crowd maze again. But this time the maze structure will be quite different. There will be only three identical rooms. Let's call them 1, 2, and 3. The player enters room 1 from the arena. All exits from 1 take the player to the room he's already in, except for one direction, which leads to room 2. All exits from room 2 take the player back to 1 again, except for one exit which leads to room 3. Again, all exits from room 3 lead back to room 1, save for one which leads outside the Coliseum. Without a clue – and remember, object dropping isn't permitted – the player won't have a chance. So we let Caesar tell the player how to leave: "Use the NEW exit," says Caesar. This may puzzle the player – it's supposed to. But the correct exit sequence outside the arena is N (to room 2), E (to room 3) and W (to leave the Coliseum). So we've told the player the solution.

The player now has three treasures – time for the 'endgame', or final puzzle, which is always nice to

have in a game. Entering the Senate in daytime, or with less than three treasures, results in assassination by senators waiting to kill Caesar (Ides of March, and all that). Entering after dark, and with three treasures, produces the Senator, who will pull a knife on the player and try to kill him. The only solution is to throw the torch into a convenient swimming pool, plunging the room into darkness (memo – hint that to player), and kill the Senator in the confusion. End of game, and end of plot!

Here's some specimen dialogue to set the scene:

You're in your house, which is poor but comfortable. To the west lies a street.

INV You are carrying: Nothing

W You are in a long east-west street. To the north lies the Senate, and to the south is a small shop.

MOVE S You are in an old shop, with its exit northwards. A shopkeeper is keeping his eye on you. There is a brazier of glowing coals here. A military helmet lies nearby.

GET HELMET The shopkeeper won't let you!

KILL SHOPKEEPER You can't do that!

GET BRAZIER You can't take that!

BOTHER (You can probably guess the response to this. . .)

4.3 More on states; introducing properties

Having created our plot – with almost no reference as to how we shall program it – let's consider whether the system outlined in Part 3 is adequate to deal with it. The answer is yes, but only just; and we can do much better. The previous system involved some ugly programming; the price to be paid for improving it will be the creation of a separate program to ensure that data gets into memory correctly.

To see what new features it would be useful to include, let's examine which parts of 'ROMAN' look hard to write. For the time being we'll restrict our attention to objects and rooms. Consider first the long

animal chain: trap to mouse to cat to chicken to priest.

Part of the chain is straightforward and demands unqualified responses: 'GET CAT' either produces a message saying the cat can't be picked up, or a message about the cat jumping into your arms and eating the mouse. The state concept is of no help, though it is for the trap problem. The result of laying the trap is an 'invisible' exit program on all exits from the farm: if the trap is in the room and if the mouse is in room zero (destroyed) and if the state of the mouse is zero, move the mouse to the room and set the mouse's state to 1 so it won't ever be placed again. (With some thought, you'll see that one of these tests is unnecessary: which, and why?) We can also use states to monitor the priest and lion. Their initial states are 0; when they and the player are in the same room, we set their states to 1 in the pre-program. Then in the post-program, we look to see if their states are 1. If they are, the player didn't deal with them properly, so we deliver the appropriate message and kill the player.

So things look good for this set of problems. States seem to handle the problems easily and neatly. It is the factors raised by the solutions that yield difficulties. 'KILL' is a verb, useful for the chicken. What about 'KILL GLADIUS'? Or 'EAT GLADIUS'? Wouldn't it save a long series of IF statements to jot down which objects were vulnerable, or eatable.

Consider now the light problem. How do we tell the program when the sun sets, so that it makes the rooms dark? We can't use states to signify which rooms are dark as we're using them for various other things anyway. How do we let certain objects – here the torch – be a light source under certain circumstances? Wouldn't it be useful if 'treasure' could be tagged onto some objects, for scoring purposes?

Perhaps the most obvious shortcoming of the states technique is demonstrated by attempts to handle the disappearance of dropped objects in the maze rooms. We could set each room's state at 10, say, as no other room would get that big, and lose objects dropped in a room with state 10, but it's not very neat. Or we could lose objects only in rooms with certain numbers, which isn't neat either.

What all this is leading up to is a plea for the addition

of some properties, different ones for objects and for rooms. Unlike a state, which is a single number from 0 upwards as high as we need, a property is like an on-off switch. It's either on or off for each object, like being a light source, for example. The gladius is never a light source, so never possesses the property 'light source'. The torch may sometimes be a light source (its state may also be different, to signal a different description, but we'll come to that later). So we can think of a property as being a TRUE/FALSE flag, or a 1/0, with a given object or room perhaps having several properties TRUE (or set) and several FALSE (or unset).

This makes the programming a lot cleaner, and hence more reliable. 'EAT object' is then handled by 'If object is not EATABLE, say so and quit. Else check if held, etc., etc.' 'DROP object' will include a line 'If room has property DROPLOSE, destroy the object and say it was lost in the crowd'. Describing the player's room includes 'If room does not have property LIGHT, and none of the objects in the room or in the player's possession has the property LIGHT SOURCE, print "It is pitch dark" and quit'.

The concept of properties really does make life easier. Types of property will vary from game to game, but some, such as light sources, for example, will probably remain constant from game to game. One very convenient property for rooms will turn out to be 'VISITED'. We'll really only scratch the surface of the use of properties in this book, but you'll find you keep thinking of uses for them.

4.4 A better message system

Now let's look at our message system. 'System?' I can hear you say, 'What system? We just print a character string and that's all!' There lies the problem – or rather two problems. The easier is that we must now extend the length of messages over 255 characters. The harder problem is that the printing of messages is an unintelligent machine response – 'print message 30' does just that and no more.

Frequently in Part 3 we wished to print a particular message and then print another under certain different circumstances (e.g. 'which you are wearing' when the state of the shield was 1. . .or a description of the locked door only for a particular room state. In

'ROMAN' there'll be many more examples of this. When describing the torch, there will be a choice of three descriptions (a piece of wood, a lit torch, a black stump) depending on its state. And more than just descriptions are involved, especially if we want to save space. When the player tries to get past the guards with the helmet, we'll need two responses depending on whether the player is holding the helmet (and one other for successfully passing the guards, of course). Either 'The guards see you are not a soldier. They bar your way' for no helmet, or 'The guards see your helmet, but realise you are not a soldier. They bar your way' if the helmet is carried but not worn. The first part of the message varies, but the second sentence remains the same. It would be nice if the message system could handle this without our having to store two identical second sentences within the machine, or to have to ask twice for the same sentence.

The ugly way of doing all this was used in Part 3: IF statements as and when necessary. As game size grows, these become more and more clumsy; slower and slower; and more and more space-occupying. So is there a better way to produce messages?

Here we will introduce a semi-intelligent message system. A message will now be stored as a collection of characters – possibly none – and a collection of switches, again possibly none. Each switch is the number of another message. A call to the message system to print, say, message 19 will then do the following: first, the characters of message 19 will be printed out if there are any. If there are no switches present, the system quits. If switches are present, however, the system then examines the value of Z%, which we shall use here as a switching parameter. If Z% is zero, the system reads the first (really zeroth) switch, and 'switches' to printing out that (new) message – which may well have its own switches, and so on. If Z% is one, the second switch is used; if Z% is 2, the third switch is used, and so on. If Z% is bigger than the supplied number of switches, the last (highest) one is used.

This sounds complicated, but it isn't. The idea is to let the message system handle all the messy details and so ease the programming burden.

Some examples may be useful here. Think again

about describing the torch. We'll let its describing message be 20, say. The appropriate subset of the messages then looks like this (with message numbers deliberately spread out to demonstrate that there's no necessity for switching only to nearby message numbers):

20 (message is null, or empty)

switches 71, 82, 97
. . ..

71 A piece of wood

(no switches)
. . ..

82 A burning torch

(no switches)
. . ..

97 A black stump

(no switches)

With Z% set to the state of the torch – which the describing system will do automatically, by the way – printing message 20 gives: first, no message; second, a switch to 71 if Z% = 0, 82 if Z% = 1, or 97 if Z% > = 2; third, the appropriate message is printed out; and fourth, the system quits as none of the latter three messages has any switches after them.

Hence merely setting the state of the wood is all the writer has to do to handle all future descriptions of the wood! Let's try the example of the guards. We set up messages 83 and 84 as:

83 The guards see you are not a soldier.

switch 85

84 The guards see your helmet, but realise you are not a soldier.

switch 85

85 They bar your way.

(no switches)

In the program, printing message 83 or 84 produces both the correct first line and the correct second line (message 85); yet only one printing instruction is necessary, and we only have to store message 85 once. We'll see many more examples when we come to the actual programming.

In Part 3 we identified the need for two types of object description: a short description ('A shield') for use when the player is holding the object, and a long description ('There is a sturdy shield on its side here.') for use when the object is not held by the player. Here we cannot switch these messages (i.e. determine which message to give the player) on the basis of the object's state, because merely taking the object doesn't usually change its state. And it would be impractical to switch on the property, TAKEN, as it would have to be set and reset during GET, DROP and THROW. Instead we must store two message numbers with each object, one for the short and one for the long description, and choose which one depending on whether the object's 'room' is 1 (i.e. the player). These descriptions may well switch within the message system (as is the case with the torch, which will need a total of 8 messages: a (null) short, switching to one of three short messages, and another four for the long descriptions.)

Rooms too will require two descriptions. Hitherto our rooms had only one description, but it is boring and unnecessary always to read a lengthy description no matter how many times one revisits a room. A message like "You're in the house" is very nice to get once in a while! So we shall provide each room with a short and a long description and choose which to use on the basis of whether the room property VISITED is set. Requesting a long description with LOOK will then only involve de-setting VISITED on the current room before describing it.

These improvements complete those at the conceptual level. Now we shall turn to improvements at the programming level.

4.5 Direct memory access

In the games and methods we have studied so far, information was stored either in arrays or data

statements. These are traditional BASIC ways of holding dynamic and static data respectively. For many purposes, they are ideal; but for use in games whose storage may well approach the limits of the computer, the overheads involved outweigh the convenience. In Part 2 we saw how to pack a great deal of information into a single array element, but more efficient ways were promised.

These efficient methods discard the use of BASIC variables completely, and address the computer memory directly. Such methods are, however, fraught with danger. When we create a variable test% we know that BASIC will choose a unique location in memory for it; that it won't place it in the middle of the screen memory, or the operating system, or your program. Furthermore, you never need to know exactly where in memory test% lives; that's BASIC's problem. But if you decide to put a value somewhere in the machine yourself, you suddenly have to worry about all these things. Fortunately, the operating system and the BASIC on the BBC Micro and Electron are sufficiently intelligent to let us access memory directly without disasters.

How one accesses memory directly is described in the User Guides: Chapter 39 for the BBC Micro and Chapter 23 for the Electron.

Three new symbols are introduced: ?, !, and $. Let's look at '?' first. If, in some BASIC expression, we write ?8000, we mean the value held in byte number 8000 in the computer (there being some 32000 bytes available, roughly). So 'X=?8000' would set the variable X to the value held in byte 8000, while '?8000=Y' would put the value of Y into byte number 8000. (Thus the '?' operator acts like both PEEK and POKE on other computers.)

That would be the end of the story, but for one thing. A byte in the computer can only contain an integer from zero to 255 inclusive (for example, an ASCII character). So we can't store 497, 3.62, or -20 in a single byte, because they are respectively too big, not an integer, and negative. For most purposes in Adventure games, this is irrelevant. We tend to work with integers anyway, and small integers at that.

There will be times when we need to handle bigger

numbers (our messages may number over 255, for example, even if neither objects nor rooms do). A BASIC integer, as we saw, can be far bigger than 255. It achieves this by occupying not one, but four bytes, one after the other in the machine. Accordingly, Acorn have supplied the '!' (pronounced 'pling') operator. Thus !8000 refers to the (possibly very large) 4-byte integer stored at bytes 8000, 8001, 8002, and 8003. The actual way it's stored is a little complicated, involving hexadecimal notation and fortunately doesn't matter for our purposes. Anyway, 'X=!8000' and '!8000=Y' are equally permissible BASIC statements. They set X to the integer at 8000, or the integer at 8000 to Y, respectively.

The '?' and '!' operators allow us to place and retrieve information directly in memory without having to pay the space overheads required by DIM statements. A table of 100 integers could easily be set up by statements like 'FOR I% = 0 TO 100: ?(8000+I%) = I%: NEXT', for example, and the table occupies exactly 100 bytes (well, 101 really!). The equivalent BASIC DIM statement would need 412 bytes, even in immediate mode: net saving, 75%. So frequently is this operation performed, that Acorn allowed '?' and '!' to act as partial adding operators: I%?8000 is a synonym for ?(8000+I%). Whether you use this contraction – which has a few limitations on it – is a matter of personal preference or a desperate need to save 3 bytes of program!

We can also access strings directly, a thing PEEK and POKE don't easily allow, so we are well off with Acorn! Just as !8000 meant 'the integer starting at byte 8000', so $8000 means 'the string of characters starting at byte 8000'. It's worth seeing just what happens when we write '$8000 = "FRED"' in a BASIC program, as we'll need the details later. Try typing this, and then request the contents of memory by 'FOR I% = 8000 TO 8004: PRINT I%, ?I%: NEXT'. What we find is:

```
byte        contents        ASCII character
8000        70              F
8001        82              R
8002        69              E
8003        68              D
8004        13              return
```

(you won't see the right-hand column!) Each character in the string is converted to its ASCII value and stored sequentially in memory. The end of the string is noted by a 13, or carriage return.

Once $8000 is in place, we can do all the things with it that one would normally do with strings (e.g. LEFT$, MID$, LEN, etc.). Printing $8001 would give "RED", for instance, as the string starting at $8001 has no "F" on it. Printing LEN($8002) gives "2" for the same reason. Rewriting '$8000 = "GA"' would overwrite bytes 8000 to 8002, giving:

byte	contents	ASCII character
8000	71	G
8001	65	A
8002	13	return
8003	68	D
8004	13	return

which, you'll see, hasn't altered 8003 or 8004 at all. Play with this for a little if you're unused to it, and you'll see that it's really rather flexible.

We can then make an array of – say – 100 four-character strings by 'FOR I% = 1 TO 500 STEP 5: $I% = "FRED": NEXT' which takes up 505 bytes. The equivalent 'DIM A$(100): FOR I% = 1 TO 100: A$(I%) = "FRED": NEXT' takes up 812 bytes; again, direct storage makes a large saving!

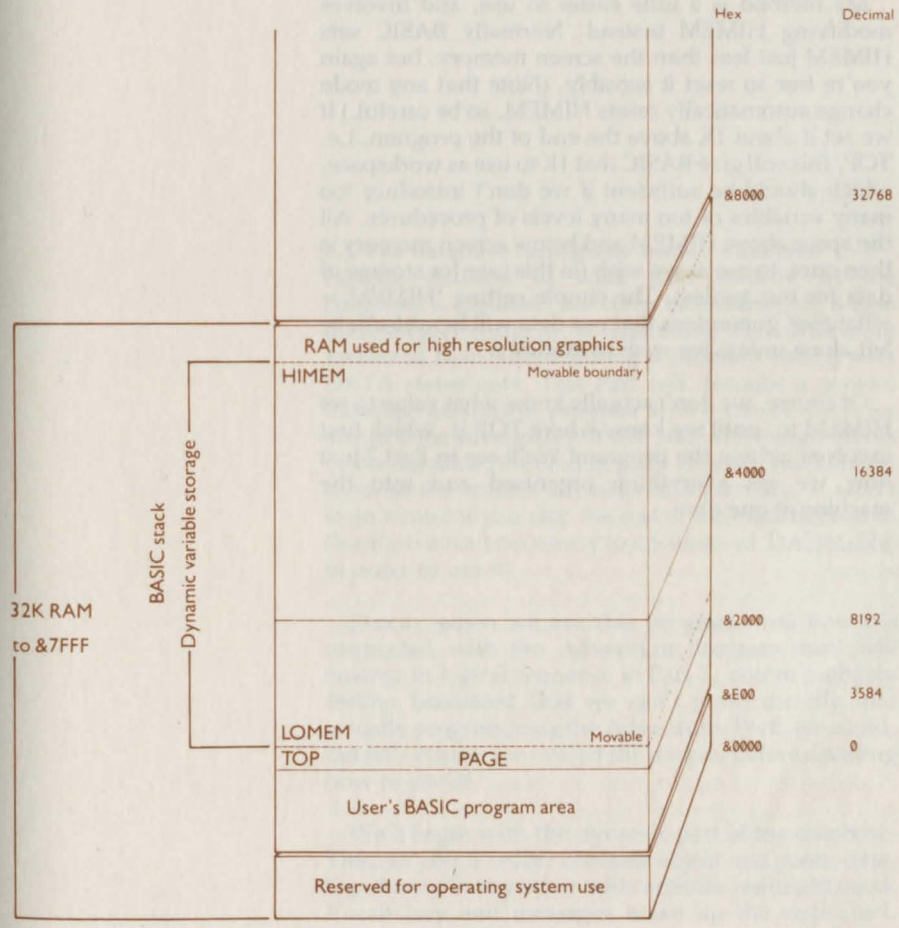
4.6 Use of direct memory access for database handling

Having seen that we can store numbers directly in memory, and by so doing save a great deal of space, we need to look at how we can do so safely. For this, we utilise the 'pseudo-variables' LOMEM and HIMEM in the computer. These relate to where in the machine your program and its workspace live. Were we to produce a simplified map of the 32K of our machine, it would look something like this, working up the memory as we go down the page:

0: Operating system
PAGE: Beginning of your BASIC program
TOP: End of your BASIC program
LOMEM: Beginning of the work-space for your BASIC program
HIMEM: End of the work-space for your BASIC program

Beginning of screen memory

In this list, I've labelled various locations. Some you've probably used before, whereas LOMEM and HIMEM may be new. In sequence, we find that the first few K of the machine are used by the operating system. After this, your program area – marked by PAGE – begins. You can set PAGE to anything that makes sense; your program will then start from that byte in the machine. TOP, on the other hand, you have no control over: as you modify your program, so its value changes. If your program gets longer, so TOP goes up.



Normally, BASIC uses all the rest of the machine apart from the screen memory for its workspace. Thus it sensibly sets LOMEM to just above TOP. However, you can modify LOMEM if you wish (but not once any variables have been set, else BASIC won't be able to find them again!) So if you want to put some directly accessible data in memory, you could put it just above your program and reset LOMEM. The snag with this method – if your programming ability is like mine, anyway – is that by the time you've got the bugs out of your program it's rather longer than it started out, and the precious data probably disappeared in the middle of some editing.

My method is a little easier to use, and involves modifying HIMEM instead. Normally BASIC sets HIMEM just less than the screen memory, but again you're free to reset it sensibly. (Note that any mode change automatically resets HIMEM, so be careful.) If we set it about 1K above the end of the program, i.e. TOP, this will give BASIC that 1K to use as workspace, which should be sufficient if we don't introduce too many variables or too many levels of procedures. All the space above HIMEM and below screen memory is then ours, to use as we wish (in this case for storage of data for our games). The simple setting 'HIMEM = whatever' guarantees that our data will henceforth be left alone unless we wish to modify it.

Of course, we don't actually know what value to set HIMEM to, until we know where TOP is, which first involves writing the program! We'll see in Part 7 just how we get everything organised and into the machine at one time.

5 INTERLUDE: AN ADVENTURE GAME DATABASE WRITING PROGRAM: 'DATAGEN'

5.1 The database format and binary numbers

Having decided to store data directly in the computer's memory, we have to ensure that it gets into memory correctly, as examining it is a little more awkward than just scanning a program listing with DATA statements. This Part will describe a general database-producing program which can be used for any of your adventures. It will take all the ingredients of the database, entered in plain English, and convert them to the correct format within memory. It won't even matter if you skip the rest of the explanations in this Part; it isn't necessary to understand 'DATAGEN' in order to use it!

Exactly when we use this program, and how it's connected with the Adventure program itself will emerge in logical sequence in Part 7. You're probably feeling frustrated that we can't jump directly into actually programming the Adventure. Well, we could, but let's ensure we can get the data in, before deciding how to use it!

We'll begin with the dynamic part of the database. This, as you'll recall, contains object and room data, together with any other odd variables we might need. Vocabulary and messages make up the static half.

Since objects are simpler than rooms, let's start with them.

We identified in Part 3 a need to store the room an object was in and a numeric variable for the object's state . In Part 4 we also created the idea of an object's properties: a collection of TRUE/FALSE flags. Finally, objects were to be accorded the numbers of two descriptive messages, one short and one long, the short one to be used when the player was holding the object and the long for when he wasn't. We shall assume that there are less than 255 rooms, that the state lies between 0 and 255, and that both message numbers are less than 255. Thus all but the properties will occupy a grand total of four bytes for each object, a great saving in space.

How are the properties to be stored? On the face of it, any eight properties, say, will occupy eight bytes. However, we can take a lesson from 'CAVES'. In that program, a pattern of zeros and ones (the structure of the room exits) was held in a single integer: 1001 meant the pattern one, zero, zero, one. To obtain every possible pattern of ones and zeros meant storing numbers from 0 to 1111. Now our single byte storage can only hold numbers up to 255. Can we improve on this? Fortunately, the answer is yes, provided that we cease working with decimal numbers and use binary numbers instead.

As far as Adventure writing is concerned, there are two equally acceptable attitudes towards binary numbers: ignore them, and hope that they will go away, or try to understand them. If you already understand at least vaguely how binary numbers operate – if you've ever defined new characters, for example – or if you'd prefer never to understand them, while still having a usable (but black-box-like) database system, then skip the next few paragraphs with my blessing.

To see how to cram more information into less space by using binary digits, let's return to that Part 2 example of 1001 meaning its four successive digits 1, 0, 0, and 1. We extracted each of the four digits by a slightly painful process, you'll recall. To get the hundreds digit (a zero) we divided by 100 to lose the tens and units, then MODded with 10 to lose all the higher digits.

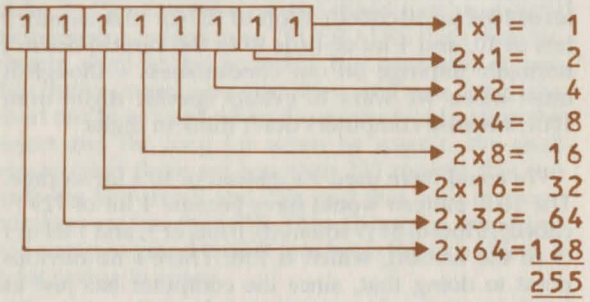
Now powers of 10 were involved all through that process: dividing by 100, MODding with 10, and so on. But there was nothing magical about using 10, except that the answers were especially acceptable to our decimal-orientated eyes. We can see the 1, 0, 0, 1 pattern when we look at 1001. The fact that it means 1 lot of 1000 (or 10 cubed), 0 lots of 100 (or 10 squared), 0 lots of 10, and 1 lot of 1 (or 10 to the zeroth) doesn't normally impinge on our conciousness – though it must when we want to extract specific digits from 1001, because computers don't think in digits.

We could have used 9's instead of 10's for storage. The 1001 pattern would have become 1 lot of 729 (9 cubed), 0 lots of 81 (9 squared), 0 lots of 9, and 1 lot of 1 (9 to the zeroth), which is 738. There's no obvious point to doing that, since the computer has just as much work to do extracting information from 9's storage as it has with 10's, and the eye can't immediately see the ones and zeros any more. However, notice that even the highest possibility, 1111, coded as 729 + 81 + 9 + 1 = 820, now only occupies 3 digits rather than 4, so we have in fact saved a digit. Though unfortunately no space has been saved: a BASIC integer takes up the same space whether it's 3 or 4 digits!

So how do we save space? The clue lies in the shrinking of 1111 (base 10) down to 820 (base 9). Suppose we continue reducing the base number and see what happens. 1, 1, 1, 1 in base 8 would be 512 (8 cubed) + 64 + 8 + 1 = 685, which is smaller still; in base 7, it becomes 400; in base 6, 259; and so on, down to base 3, 40. The least is to base 2, which gives 8 + 4 + 2 + 1 = 15 – rather a reduction on 1111, and yet containing exactly the same information! Note, though, that we could also handle room codes like 8, 4, 7, 6 in base 10 – it's just 8476 – but we can't go over 0's and 1's in base 2. This is because 2 lots of 2 squared, for example, happens to be 2 cubed, so that an entry of more than 1 would be confused by the computer when retrieving the numbers. So, use decimal if you have to store 0's to 9's; or, if space is no object, use binary to store 0's and 1's.

How many 0's and 1's can we cram into a byte – i.e. and still have a number less than 256? The answer is 8 (which is one of the reasons why the computer is an 8-bit micro, by the way). Thus 1, 1, 1, 1, 1, 1, 1, 1

becomes 2 to the 7th power + 2 to the 6th power + . . . + 2 + 1 = 255 exactly, while at the other extreme 0, 0, 0, 0, 0, 0, 0, 0 yields 0. Hence we can put 8 binary properties into a single byte! We shall discuss getting them out again later.



We can now specify the object database format. Each object will occupy five bytes in memory.The first byte will be the object's state; the second will hold its eight properties, with zeros meaning 'not set' and ones meaning 'set'; the third byte holds the room of the object; the fourth the message number of the object's short description; and the fifth byte the number of its long description. Each group of five bytes is strung out one after the other, beginning with object zero. This one doesn't exist, but it's quite useful to have it around, just as there are uses for room zero. The setup is shown diagrammatically below, with the list being as long as necessary:

```
state   props    room   short    long I  short   props    room  ...
  A       A       A       A       A   I   A       A       A
------------ object 0 ---------------I------ object 1 --------...

        (Each 'A' represents a byte in memory)
```

You may still be feeling that all this complication is unnecessary. Well, dimensioning a state array, 8 property arrays, a room and two message arrays for 60 objects will take up nearly 2900 bytes of your machine – about 9% of it. Direct storage of the same 60 objects takes up only 305 bytes, about one-tenth of the dimensioning method; but we have to pay for this by some program to get at the numbers.

What of the rooms part of the database? Each room, too, has a state, 8 properties, and two descriptions. But in addition, we need to know about exits – all 10 of

them: N, NE, E, SE, S, SW, W, NW, UP and DOWN. You can easily include others, such as IN and OUT, if you wish – even exits which the player can't use but the program can! (Did you ever want to have a monster following the player through magical exits? Give the monster a spare direction to use, and don't let the player get at it!) For each exit, we shall provide 2 bytes of information: first the room to which the exit leads (zero in the case of no exit), and second the number of an exit program to be obeyed before going through the exit.

With this exit program number, which is normally zero indicating no program, we crack the messy series of tests we had to make in 'MINI' ("are we going north and in room 3?") All movement becomes a single program: (a) look up exit; (b) if zero, quit; (c) check for exit program, obeying it if necessary; (d) if exit is still permissible, move the player. Not only does this look far neater and require far less storage, it's also far more structured and thus infinitely more likely to work correctly in your program.

The rooms database will thus look like this, beginning also with room zero. Each room occupies 24 bytes (state, properties, 2 messages, and 10 directions times 2 bytes):

```
state props Nexit Nprog NEexit NEprog ... Dexit Dprog short long I state
  A     A     A     A     A      A          A     A     A     A  I   A
------------------------- room 0 ------------------------------I ----
```

In general, the rooms database would follow on directly after the last object in the objects database. After the last room you could leave space for any other odd variables you might need: the battery life of a lamp, or whatever. Sometimes such values can be tucked into unused state areas, etc., as we'll see in 'ROMAN'.

After the rooms will come the static database, consisting of vocabulary and messages. The vocabulary begins with the 'doing' words (all verbs in our parlance). Each verb occupies 6 bytes: four for its first 4 characters, padded out with spaces for short verbs like 'E', then one byte for the verb number and one for the verb type. All vocabulary will be in lower case. This is because we're going to let our player type in upper or lower case, convert his input in the program to lower case, and compare it with the database. So we have:

```
     verb      number type I  verb      number type I ...
A  A  A  A        A      A   I       A  A  A  A       A    A   I ...
--------- verb 1 ----------I------------- verb 2 ----------I
```

as far as necessary.

The 'things' part of the vocabulary follows on directly. Each thing occupies 5 bytes: four for the vocabulary itself, and one for the object number. Hence:

```
     thing     number  I       thing     number  I ...
A  A  A  A       A     I   A  A  A  A       A    I ...
------ thing 1 -------I------ thing 2 ---------I ...
```

also as far as necessary. The 'specials' have the same format, and follow on after the objects:

```
     special    number  I     special    number  I ...
  A  A  A  A      A     I   A  A  A  A      A    I ...
------ special 1 -------I------ special 2 -------I ...
```

This only leaves the messages. The messages will follow on directly after the special vocabulary. Each message consists of three sections. The first is a single byte which contains both the number of switches in the message, and the number of lines – we'll see how later. The second section consists of each line of the message, stored using the $ symbol as in Part 4. The third section – which may not be there – is a collection of pairs of bytes, each of which holds a message switch. These have to be a pair, because there may well be more than 255 messages; in that case, a switch to a number bigger than 255 couldn't be held in one byte, and it's easier to set up two from the start. Again, we'll come to exactly how we do that; understanding it isn't a condition of using it.

Thus the message layout looks like this, where 's/l' stands for 'switch/lines':

```
s/l  line 1 ...    line 2 ...     sw0  sw1  sw2  I  s/l ...
 A   AAAAA..A 13   AAAAA..A 13 .. AA   AA   AA   I   A  ...
----------- message 1 -------------------------I---- message 2 ---
```

5.2 The driving program

We can now write a straightforward program to read in data from keyboard and convert it into suitable direct memory format, and put it in the right place in memory. This section details the outer core of that program, 'DATAGEN':


```
 10DIMDIR$(10)
 20DATA N,NE,E,SE,S,SW,W,NW,U,D
 30NO=0:NR=0:NV=0:NT=0:NS=0:NM=0
 40U%=&FFFF0000:L%=&FFFF
 50FORI%=1TO10:READ DIR$(I%):NEXT
 60INPUT"TIMELAG?"TT
 90INPUT"OBJECT,ROOM,VERB,THING,SPECIAL,MESSAGE?"X$
100IFX$="" END
110ON INSTR("ORVTSM",X$) GOTO 400,1000,1500,2000,2500,3500
```

Line 10 sets up a 10-element array to hold the 10 possible directions. These are listed in the data statement at line 20, and read in at line 50. Line 30 initialises the number of objects (NO), rooms (NR), verbs (NV), things (NT), specials (NS) and messages (NM) to zero. The mysterious line 40 sets U% and L% to hexadecimal values used in inserting message switches. Ignore them for now; Appendix 2 explains a little of how hexadecimal works. Line 60 asks for a timelag between each set of input in centiseconds. The idea is that you may want a moment to study the output from each object, or whatever, to see if it's correct. Hence the breathing space. At line 90 the user inputs the letter 'O' for object, 'R' for room, etc., to specify which piece of the database will be input next. It is customary for me to input them in the order listed. Inputting a blank line ends the program (line 100). Line 110 sends the program off to the appropriate area to handle the database section required.

5.3 Object, room and vocabulary storage

Now let's first see how we feed in numbers for the object and room storage, (the dynamic part of the database) handled at 400 and 1000 respectively. A look at the appendices may be useful here. First the object storage:

```
 399REM OBJECT
 400INPUT "Value of o%=?"X$:o%=EVAL(X$)
 405REM  *EXEC OBJECTS
 410REPEAT INPUT'"OBJECT NUMBER?"O:IFO=0END
 420P%=o%+5*O:NO=NO+1
 430INPUT"STATE?"I%:?P%=I%
 440J%=0
 450REPEAT INPUT"PROPERTY?"K%:IFK%<OORK%>7 UNTIL TRUE ELSE
J%=J%+2^K%:UNTIL FALSE
 470P%?1=J%
 480INPUT"ROOM?"J%:P%?2=J%:P%=P%+3
 490INPUT"SHORT LABEL?"J%:?P%=J%:P%=P%+1
 500INPUT"LONG LABEL?"J%:?P%=J%:P%=P%+1
 510PRINT"P%=";~P%,P%
 520PRINT"CHECK: ";:P%=o%+5*O:FOR I%=0 TO 4:PRINTP%?I%;:NEXT
 530PRINT'"NO= ";NO
 540PROCWAIT
 550UNTIL FALSE
```

Explanations:

400: Input the value of where in memory object zero will be stored, here called o%. The EVAL is to enable you to input it in hexadecimal if you wish. You may gulp at that, but there are advantages to this even if you don't understand hexadecimal! Don't worry that we don't know where to put the objects yet; we have to write the main program before we know that.

405: This is a rather strange line. Ignore it (as it's REMmed) unless you possess a word-processing package. If so, you can prepare all your input for these programs by word-processor, and then*EXEC it in here.

410: Begin unending repeat loop. Input object number, and quit if it's zero or blank.

420: Set P% to the place in memory where object O will begin, and increase the object count.

430: Input state. This goes exactly at byte P%.

440: Begin property setting by making J% – which will hold the 8 properties – to zero, i.e. none set.

450: Keep asking for properties to be set until one out of the range 0 to 7 is input (I use 10 as a terminator). If the property is in range, add property K% to the J% total. Note the power of 2, just like we used powers of 10 before. So don't repeat a property, or chaos will ensue.

470: Finally set the byte one after P% to the property count J%.

480: Object's room goes at 2 after P%, then increase P% for the message labels.

490-500: Get short and long labels, and insert in memory. Increase P% to point to next byte each time.

510: Let the user know where in memory we've reached, printing P% in both hexadecimal and decimal. I repeat that you don't need to understand hexadecimal, but occasionally knowing values in hexadecimal is important, as we'll see. The use of the tilde (~) is mentioned in Appendix 2.

520-530: Take no chances – print out the contents of the 5 bytes whose values we just set, and the number of objects we've done.

540-550: Wait the prescribed time, then end the loop.

The rooms portion is well-nigh identical, except for the addition of the exits and their program numbers. These latter are part of the static database, since their contents never change.

```
999REM ROOM
```



```
1000INPUT"Value of r%?"X$:r%=EVAL(X$)
1005REM *EXEC ROOMS
1010REPEAT INPUT'"ROOM NUMBER?"R:IF R=0END
1020P%=r%+24*R:NR=NR+1
1030INPUT"STATE?"I%:?P%=I%
1040J%=0
1050REPEAT INPUT"PROPERTY?"K%:IFK%<00RK%>7 UNTIL TRUE ELSE
J%=J%+2^K%:UNTIL FALSE
1070P%?1=J%
1080FORI%=2TO21:P%?I%=0:NEXT
1090REPEAT INPUT"DIRECTION?"X$
1100IF X$<>"" I%=0:REPEATI%=I%+1:UNTILX$=DIR$(I%):INPUT"ROOM AND
CODE?"R1,R2:P%?(2*I%)=R1:P%?(2*I%+1)=R2
1120UNTIL X$=""
1130P%=P%+22:INPUT"SHORT LABEL?"J%:?P%=J%:P%=P%+1
1140INPUT"LONG LABEL?"J%:?P%=J%:P%=P%+1
1150PRINT"P%=";~P%,P%
1160PRINT"CHECK: ";:P%=r%+24*R:FOR I%=0 TO 23:PRINT;P%?I%;:NEXT
1170PRINT'"NR=";NR
1180PROCWAIT
1190UNTIL FALSE
```

Explanations, ignoring the identical portions:

1000: Set r% for rooms.

1010: Quit when zero room number entered.

1020-1070: State and properties.

1080: Set the exits-and-exitprogs section all to zero (no exits, no programs).

1090: Repeatedly input a direction that player may leave this room by (N, NE, etc.) There is no need to do these in any specific order. Terminate by a simple carriage return; such an entry will be ignored by the next line.

1100: Find which direction it was, so I% has the value 1 to 10. The program doesn't check for an error here, as it'll tell you if you mistyped by failing! Then input exit and program number in that direction. Notice the comma, so you can type '17, 3' to mean room 17, exit program 3, all on one line. Then set the appropriate two bytes of memory.

1120: Keep going!

1130-1190: Pretty much as with objects, save that a lot of numbers, all scrunched up, are printed out. You could put spaces between them if you wished.

Just to recap, then, when the time comes to type in the objects and rooms database, we choose the appropriate option and steadily, for each object/room, type in the information required, terminating each piece as I've indicated. It is wise to jot down the final value of P% after each set of data in both hexadecimal and decimal for use later.

The remainder of the database is static, and begins with the vocabulary input. Verbs are first, at 1500:

```
1499REM VERBS
1500INPUT"Value of v%?"X$:v%=EVAL(X$):P%=v%
1505REM *EXEC VERBS
1510REPEAT INPUT"VERB? "Z$:IF Z$=""END
1520NV=NV+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
1530IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
1540INPUT"PROG LABEL, CODE? " P,C
1550$P%=Z$:P%=P%+4:?P%=P:P%?1=C:P%=P%+2
1560PRINT~P%,P%
1570PRINTNV:PROCWAIT:UNTIL FALSE
```

Explanations are fairly obvious. We set v% to be the byte in which the first verb is placed. A loop is entered at 1510, terminating with a null input. Line 1520 shortens the verb to 4 letters, and 1530 pads it out to 4 with spaces. Line 1540 accepts the program label – i.e. which command the verb really is – and its type, which can come in on one line with a comma separator if you wish. At 1550 we first store the verb in $P%, which uses 5 bytes (4 for the verb, one for carriage return), then overwrite the carriage return with the program label. Finally, the code is set, and P% modified to point to the next verb's position. This is printed at 1560 in hexadecimal and decimal, and after a wait at 1570, the loop continues.

Things and special words are almost identical:

```
1999REM THINGS
2000INPUT"Value of t%?"X$:t%=EVAL(X$):P%=t%
2005REM *EXEC THINGS
2010REPEAT:INPUT"THING? "Z$:IF Z$=""END
2020NT=NT+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
2030IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
2040INPUT"LABEL? " L
2050$P%=Z$:P%=P%+4:?P%=L:P%=P%+1
2060PRINT~P%,P%
2070PRINTNT:PROCWAIT:UNTIL FALSE
2499REM SPECIALS
2500INPUT"Value of s%?"X$:s%=EVAL(X$):P%=s%
2505REM *EXEC SPECIAL
2510REPEAT:INPUT"SPECIAL? "Z$:IF Z$=""END
2520NS=NS+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
2530IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
2540INPUT"LABEL? " L
2550$P%=Z$:P%=P%+4:?P%=L:P%=P%+1
2560PRINT~P%,P%
2570PRINT NS:PROCWAIT:UNTIL FALSE
```

No explanation is needed for either, as the only difference from verbs is the one less byte used in storage. Typically, the verbs are stored immediately after rooms, with things and specials immediately after them. This wastes no space – but it does mean a

bit of moving if you find you omitted some particular piece of vocabulary!

5.4 Message and switching storage

The only remaining part of 'DATAGEN' concerns the messages, which typically take up most of the database. Here's the program section. Most of the going is easy, but life gets tough at the end and you may want to skip that part unless you enjoy hexadecimal!

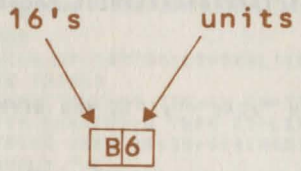
```
3499REM MESSAGE
3500INPUT"START FROM NUMBER, AND WHERE?"III,MM$:MMM%=EVAL(MM$):NM= III:
P%=MMM%
3505REM *EXEC MESSAGE
3510REPEAT:INPUT "NO. OF SWITCHES, MESSAGE NUMBER?"S,MNO
3520IF MNO <>NM+1 PRINTMNO,NM: STOP
3530IF S>15 PRINT"TOO BIG!":END
3540?P%=16*S:Q%=P%:P%=P%+1:LIN=1
3550REPEAT PRINT"LINE ";LIN
3560INPUT LINE Z$
3570IFZ$=""ANDLIN=1THEN$P%=Z$:P%=P%+LEN(Z$)+1:UNTIL TRUE ELSE IF Z$=""
UNTIL TRUE ELSE LIN=LIN+1:$P%=Z$:P%=P%+LEN(Z$)+1:UNTIL FALSE
3620?Q%=?Q%+LIN-1:NM=NM+1:IFS>0 FOR I%=0 TO (S-1):PRINT"SWITCH LABEL
";I%:INPUTL:PROCDP(P%,L):P%=P%+2:NEXT
3730PRINT~P%,P%:PRINT"NM= ";NM
3740REM PRINT FOR SECURITY
3750P1%=MMM%:I=III+1:IFNM=0 END
3760REPEAT IF NM=I UNTIL TRUE ELSE L=?P1%AND&0F:S=?P1%DIV16:P1%=P1%+1:FOR
J%=1TO L:P1%=P1%+LEN($P1%)+1:NEXT:I=I+1:P1%=P1%+2*S:UNTIL FALSE
3810L=?P1%AND&0F:S=?P1%DIV16:P1%=P1%+1
3820FORJ%=1TOL
3830Z$=$P1%
3910PRINTZ$:P1%=P1%+LEN(Z$)+1:NEXT
3920IFS>0 FORI%=0 TO (S-1):PRINT"SWITCH ";I%;"=";!P1% AND &FFFF:
P1%=P1%+2:NEXT
3950PROCWAIT:UNTIL FALSE
```

This certainly needs explaining. The first thing to realise is that you are exceedingly unlikely to type in all the messages without making an error somewhere – there is just too much text to expect otherwise. So line 3500 allows for the occasional error by letting the user specify which message and where in memory the messages will start from. In other words, you can end the program after an error, and restart from the end of the last correct entry. Line 3500 then sets P%, (after evaluation, in case of hexadecimal) to the byte where that message is to start. Line 3505 gives word-processors their chance, and can be ignored otherwise.

Line 3510 begins the loop proper. It requests the number of switches for the new message, and its number. This latter number is a security precaution

that you didn't miss any numbers; I always miss one somewhere. If MNO doesn't match with NM + 1, 3520 terminates the program. Line 3530 objects if more than 15 switches are mentioned. Line 3540 stores the number of switches in the 'top nibble' of ?P%. This is a piece of machine-coder's jargon, I'm afraid, but it's convenient. You'll remember that we want to store both the number of switches and the number of lines in the first byte of a message. Either can range from zero to 15. Now if we were using decimal numbers and had two digits available, we could store two numbers between zero and 9. One would be the tens digit, one the units. We could extract either by the methods in Part 2.

Well, should we choose to work to base 16 (hexadecimal), we could instead store two numbers up to 15. One would be the "16's" digit, and one the units digit. Each of these 'digits', to base 16, is called a 'nibble'. So in 3540, we set 16 times the number of switches into ?P%. We can't put in the number of lines yet, as neither we nor the program knows how many there are. At the moment, that number is implicitly zero (why?).



Line 3540 continues by jotting down in Q% the position of this byte, so that we can return later, moving P% to point to the next byte, and setting a line counter. Lines 3550-3560 start a REPEAT loop, and input a whole line of message (note the use of INPUT LINE to allow commas, etc., in the input). Line 3570 checks whether there were no characters on the line and it was the first one. In that case the message is null, and presumably followed by switches, so we proceed to the switch section by ending the loop, after putting a carriage return at P% and moving P% on one byte. If this wasn't the case, we again check for no input, only this time it's not on the first line and so we can just end the loop directly – the carriage return will already be there courtesy of the previous line.

If we're still here, we must have had some text on that line. So we increment LIN, store the string input at P%, and move P% on to its new position. We then go round the loop of input again.

Line 3620, when reached, marks the end of the lines in the message. We now know how many lines there were: LIN, to be precise. So we tuck LIN in as the 'units nibble' at the beginning of the message (Q%), and increment the message count. If there were no switches, that's finished that message altogether.

If there were switches – i.e. S > 0 – then we ask for them in turn. Line 3620 executes a procedure whose name is supposed to resemble 'Double Poke' to put the switch label L into two bytes of memory, followed by incrementing P% by two. We'll look at this in a moment. Lines 3740 to 3920 are because I am a coward. I prefer to see that the message has been entered correctly by getting the machine to print it all out again, switches and all. Line 3750 starts at the first message typed in this session, using P1% as a counter. If we have reached the current message, we end this REPEAT loop. Otherwise,we calculate the number of lines and switches in the message we've reached so far. The DIVving by 16 should be obvious – it gets the top nibble – but the 'AND &0F' to get the units may look odd. Why not just MOD with 16? The only reason is that ANDing with &0F (15 in decimal) achieves the same result but quicker (see the Appendices). For now, trust that it works, and if you're unhappy, change it!

Next, still at 3760, we move through the L messages; even if L is zero, there'll be one carriage return, so the FOR loop will work correctly. Then, we jump over enough bytes to handle the switches, and go round again.

Eventually we reach the message we're testing, at 3810. Again we read in switches and lines into S and L. 3820-3910 then print out the L lines straight out of memory, and increment P1% accordingly. If there are no switches (3920) we get the next message, else we print out each switch. Some more hexadecimal has reared its ugly head here. This relates to getting a two-byte number out of memory easily. Again, take this on trust – it works. After a wait (3950) we go round again.

This leaves just the PROCDP and PROCWAIT procedures. Here they are:

```
4000DEFPROCDP(P%,J%):!P%=(!P%ANDU%)OR(J%ANDL%):ENDPROC
4200DEFPROCWAIT:TIME=0:REPEATUNTILTIME>TT:ENDPROC
```

both one-liners. PROCWAIT is simple, and we'll say no more about it. But PROCDP contains some rather splendid logic. What's it all about?

If you're still reading this part, you'll now have to delve a little further into hexadecimal numbers and their storage. Acorn, you'll remember, gave us the '?' operator to catch a one-byte number, and the '!' operator to catch a four-byte number. But the switches we are using generally need more than one byte (though not in 'ROMAN') and less than four. In fact, exactly two. Had Acorn given us another operator to grab or load a two-byte number, all we would have had to do was use it. But we haven't got one, and must make do.

To be able to store two-byte numbers, we obviously need to know how four-byte numbers are stored. Well, they're stored as if to base 256 – but units first, then 'tens' – here '256s' – then 'hundreds' – here '256 squared' – and finally 'thousands' – here '256 cubed'. In this system, the 'digits' can go from zero to 255, of course. For our purposes, the order is the most important thing. It's units in the first byte, 256s in the next, and so on. So how do we use this to put two-byte numbers in and out? Suppose we just wrote !P% = 300, for example. What would happen in the storage? To find out, we need 300 to base 256. There's one 256, and a units figure of 300 – 256 = 44. So 300 is represented as 1, 44.

Thus !P% = 300 puts into memory the following:

P%	P%+1	P%+2	P%+3
44	1	0	0

since there weren't any 'hundreds' or 'thousands' in 300. The first two bytes have done exactly what we needed; but we splatted zeros over P% + 2 and P% + 3, which was not what was required, alas.

To get round this, we need to replace 300 by a figure whose units and 'tens' make up 300, but whose 'hundreds' and 'thousands' are precisely what the

digits in P% + 2 and P% + 3 were before we started loading numbers on top of them. We could get at these by '!P%DIV (256*256)', although logical operations are faster. Look back at the pattern of U% and L% as defined in line 40. Although these are written as hexadecimal – because I don't know what they are in decimal! – each pair of characters forms a digit in hexadecimal. In fact, &FFFF is 255, 255 in hexadecimal characters. So L% is exactly 255, 255 – the biggest 'two-digit' hexadecimal number. U%, however, is 255, 255, 0, 0 – rather like 9900 in decimal. It only has 'hundreds' and 'thousands' digits, with zero 'tens' and units.

So how does line 4000 operate? First we get !P% AND U%. This ANDing removes the 'tens' and units digits, thus retaining only the two digits we wished to keep. Also, J% AND L% makes sure that we've only got the 'tens' and units digit of J% here (not necessary for this program, as that's always the case). We now OR these two together. The result has the 'thousands' and 'hundreds' digits of !P% and the 'tens' and units digits of J% – exactly as required. This then goes back into !P%.

Getting two-byte numbers back out is much easier, as 3930 showed. We can just take !P% and strip off the higher two digits by ANDing with &FFFF.

Again, my apologies for this complexity. Sometimes we just have to live with complexity. Don't worry if you didn't follow it all; it doesn't matter a bit.

That concludes the description of the program to put the data in the machine. In the next Part we'll discuss how to manipulate the data within our main Adventure program.

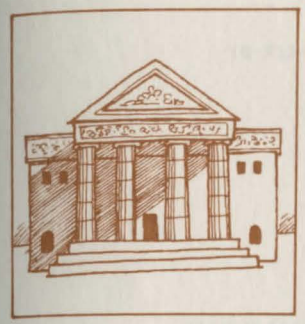
5.5 A listing of 'DATAGEN'

```
10DIMDIR$(10)
20DATA N,NE,E,SE,S,SW,W,NW,U,D
30NO=0:NR=0:NV=0:NT=0:NS=0:NM=0
40U%=&FFFF0000:L%=&FFFF
50FORI%=1TO10:READ DIR$(I%):NEXT
60INPUT"TIMELAG?"TT
90INPUT"OBJECT,ROOM,VERB,THING,SPECIAL,MESSAGE?"X$
100IFX$="" END
110ON INSTR("ORVTSM",X$) GOTO 400,1000,1500,2000,2500,3500
399REM OBJECT
400INPUT "Value of o%=?"X$:o%=EVAL(X$)
405REM  *EXEC OBJECTS
```

```
  410REPEAT INPUT'"OBJECT NUMBER?"O:IFO=0END
  420P%=o%+5*O:NO=NO+1
  430INPUT"STATE?"I%:?P%=I%
  440J%=0
  450REPEAT INPUT"PROPERTY?"K%:IFK%<OORK%>7 UNTIL TRUE ELSE J%=J%+2^K%:
UNTIL FALSE
  470P%?1=J%
  480INPUT"ROOM?"J%:P%?2=J%:P%=P%+3
  490INPUT"SHORT LABEL?"J%:?P%=J%:P%=P%+1
  500INPUT"LONG LABEL?"J%:?P%=J%:P%=P%+1
  510PRINT"P%=";~P%,P%
  520PRINT"CHECK: ";:P%=o%+5*0:FOR I%=0 TO 4:PRINTP%?I%;:NEXT
  530PRINT'"NO= ";NO
  540PROCWAIT
  550UNTIL FALSE
  999REM ROOM
 1000INPUT"Value of r%?"X$:r%=EVAL(X$)
 1005REM *EXEC ROOMS
 1010REPEAT INPUT'"ROOM NUMBER?"R:IF R=0END
 1020P%=r%+24*R:NR=NR+1
 1030INPUT"STATE?"I%:?P%=I%
 1040J%=0
 1050REPEAT INPUT"PROPERTY?"K%:IFK%<OORK%>7 UNTIL TRUE ELSE J%=J%+2^K%:
UNTIL FALSE
 1070P%?1=J%
 1080FORI%=2TO21:P%?I%=0:NEXT
 1090REPEAT INPUT"DIRECTION?"X$
 1100IF X$<>"" I%=0:REPEATI%=I%+1:UNTILX$=DIR$(I%):INPUT "ROOM AND CODE?"
R1,R2:P%?(2*I%)=R1:P%?(2*I%+1)=R2
 1120UNTIL X$=""
 1130P%=P%+22:INPUT"SHORT LABEL?"J%:?P%=J%:P%=P%+1
 1140INPUT"LONG LABEL?"J%:?P%=J%:P%=P%+1
 1150PRINT"P%=";~P%,P%
 1160PRINT"CHECK: ";:P%=r%+24*R:FOR I%=0 TO 23:PRINT;P%?I%;:NEXT
 1170PRINT'"NR=";NR
 1180PROCWAIT
 1190UNTIL FALSE
 1499REM VERBS
 1500INPUT"Value of v%?"X$:v%=EVAL(X$):P%=v%
 1505REM *EXEC VERBS
 1510REPEAT:INPUT"VERB? "Z$:IF Z$=""END
 1520NV=NV+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
 1530IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
 1540INPUT"PROG LABEL, CODE? " P,C
 1550$P%=Z$:P%=P%+4:?P%=P:P%?1=C:P%=P%+2
 1560PRINT~P%,P%
 1570PRINTNV:PROCWAIT:UNTIL FALSE
 1999REM THINGS
 2000INPUT"Value of t%?"X$:t%=EVAL(X$):P%=t%
 2005REM *EXEC THINGS
 2010REPEAT:INPUT"THING? "Z$:IF Z$=""END
 2020NT=NT+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
 2030IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
 2040INPUT"LABEL? " L
 2050$P%=Z$:P%=P%+4:?P%=L:P%=P%+1
 2060PRINT~P%,P%
 2070PRINTNT:PROCWAIT:UNTIL FALSE
 2499REM SPECIALS
 2500INPUT"Value of s%?"X$:s%=EVAL(X$):P%=s%
 2505REM *EXEC SPECIAL
 2510REPEAT:INPUT"SPECIAL? "Z$:IF Z$=""END
```



```
 2520NS=NS+1:IF LEN(Z$)>4 THEN Z$=LEFT$(Z$,4)
 2530IF LEN(Z$)<4 THEN Z$=Z$+STRING$(4-LEN(Z$)," ")
 2540INPUT"LABEL? " L
 2550$P%=Z$:P%=P%+4:?P%=L:P%=P%+1
 2560PRINT~P%,P%
 2570PRINT NS:PROCWAIT:UNTIL FALSE
 3499REM MESSAGE
 3500INPUT"START FROM NUMBER, AND WHERE?"III,MM$:MMM%=EVAL(MM$):NM= III
P%=MMM%
 3505REM *EXEC MESSAGE
 3510REPEAT:INPUT "NO. OF SWITCHES, MESSAGE NUMBER?"S,MNO
 3520IF MNO <>NM+1 PRINTMNO,NM: STOP
 3530IF S>15 PRINT"TOO BIG!":END
 3540?P%=16*S:Q%=P%:P%=P%+1:LIN=1
 3550REPEAT PRINT"LINE ";LIN
 3560INPUT LINE Z$
 3570IFZ$=""ANDLIN=1THEN$P%=Z$:P%=P%+LEN(Z$)+1:UNTIL TRUE ELSE IF Z$=""
UNTIL TRUE ELSE LIN=LIN+1:$P%=Z$:P%=P%+LEN(Z$)+1:UNTIL FALSE
 3620?Q%=?Q%+LIN-1:NM=NM+1:IFS>0 FOR I%=0 TO (S-1):PRINT"SWITCH  LABEL
I%:INPUTL:PROCDP(P%,L):P%=P%+2:NEXT
 3730PRINT~P%,P%:PRINT"NM= ";NM
 3740REM PRINT FOR SECURITY
 3750P1%=MMM%:I=III+1:IFNM=0 END
 3760REPEAT IFNM=I UNTIL TRUE ELSE L=?P1%AND&0F:S=?P1%DIV16:P1%=P1%+1:F
J%=1TO L:P1%=P1%+LEN($P1%)+1:NEXT:I=I+1:P1%=P1%+2*S:UNTIL FALSE
 3810L=?P1%AND&0F:S=?P1%DIV16:P1%=P1%+1
 3820FORJ%=1TOL
 3830Z$=$P1%
 3910PRINTZ$:P1%=P1%+LEN(Z$)+1:NEXT
 3920IFS>0 FORI%=0 TO (S-1):PRINT"SWITCH ";I%;"=";!P1% AND &FFFF:P1%=P1
NEXT
 3950PROCWAIT:UNTIL FALSE
 4000DEFPROCDP(P%,J%):!P%=(!P%ANDU%)OR(J%ANDL%):ENDPROC
 4200DEFPROCWAIT:TIME=0:REPEATUNTILTIME>TT:ENDPROC
```



6

ORGANISING AN ADVANCED ADVENTURE GAME

6.1 The object and room handling subprograms
This part will develop all the 'manipulating' routines between lines 5000 and 5999 to handle our new database. You will be delighted to hear that most of them are straightforward, and, of course, they are all usable time and again in other Adventures. This is 'bottom-up' programming, but worth it nonetheless.

The first two are fundamental. These are functions FNOL(O%) and FNRL(R%) which give the value of what I call the 'Object Label' and the 'Room Label', i.e. the byte in memory where the piece of the relevant database for that object or room begins:

```
5080DEFFNOL(O%)=O%*5+o%
5090DEFFNRL(R%)=R%*24+r%
```

These should be obvious. Object zero begins at o%, so object O% begins O% times 5 beyond that; the room label is exactly similar, only each room takes up 24 bytes and not 5. Now we know where a given object or room 'lives' in the machine, we can get hold of the information in its database.

Getting information is achieved by a collection of functions. For each object, we need to know its room, its state, and a logical TRUE/FALSE for each of the eight possible properties for that object. Here they are:

```
5100DEFFNR(O%)=?(FNOL(O%)+2)
5110DEFFNS(O%)=?FNOL(O%)
5130DEFFNP(O%,P%):LOCALI%:I%=?(FNOL(O%)+1)AND2^P%:=(I%>0)
```

Line 5100 provides FNR(O%), which returns the room which object O% is in. It merely looks up what's in the byte two beyond FNOL(O%), which is where the room is stored. Similarly, 5110 gives FNS(O%), which returns the state of O%. And 5130 gives FNP(O%,P%) which is TRUE or FALSE according as property P% of object O% is set or unset. This first evaluates the logical AND of the byte containing the properties – FNOL(O%) + 1 – with 2↑P%, which separates the single property we're interested in (see Appendix 1 for more details). If this quantity is nonzero, we return TRUE, else FALSE. By the way, powers are very slow to evaluate, so you might want to set an array POWER%(P%) equal to 2↑P% before the program ran; or a byte array. Another tip is that colons aren't needed after FNP(O%,P%); BASIC is clever enough to notice when a bracketed definition finishes. For clarity, though, I've left them in.

The room functions are identical, except there is no corresponding one for the room of an object. Their names are FNRS(R%) (Room State) and FNRP(R%,P%) (Room Property). Notice how I use short but suggestive names to save memory and execution time.

```
5310DEFFNRS(R%)=?FNRL(R%)
5320DEFFNRP(R%,P%):LOCALI%
5330I%=?(FNRL(R%)+1)AND2^P%:=(I%>0)
```

In addition to examining the contents of the object/room database, we'll also need to change them. This is done by procedures, again suggestively named. Here are the object manipulators:

```
5140DEFPROCOSP(O%,P%,I%)
5150A%=FNOL(O%):IFI%=0A%?1=A%?1AND(&FF-2^P%)ELSEA%?1=A%?1OR2^P%
5160ENDPROC
5170DEFPROCOSS(O%,I%):?FNOL(O%)=I%:ENDPROC
5200DEFPROCR(O%,R%):?(FNOL(O%)+2)=R%:ENDPROC
```

Explanations:
5140 provides PROCOSP(O%,P%,I%) (Object Set Property). This sets property P% of object O% to I%, which should therefore be 0 or 1. This is achieved by line 5150 (refer to Appendix 1 if in doubt). Either a logical AND is used to clear property P% without disturbing the others, or else a logical OR is used to

set property P%. Notice the use of the '?' operator as an addition tool here. Line 5170 gives PROCOSS(O%,I%) (Object Set State) which sets the state of object O% to I%. And line 5200 gives the very useful PROCR(O%,R%) (Room) which sets the room of object O% to be R%. I think these should be straightforward to understand.

A similar pair are required for room manipulation: PROCRSP(R%,P%,I%) (Room Set Property) and PROCRSS(R%,I%) (Room Set State). Neither should need further discussion.

```
5340DEFPROCRSP(R%,P%,I%)
5350A%=FNRL(R%)+1:IFI%=0?A%=?A%AND(&FF-2^P%)ELSE?A%=?A%OR2^P%
5360ENDPROC
5370DEFPROCRSS(R%,I%):?FNRL(R%)=I%:ENDPROC
```

6.2 The message procedure

We also need a procedure PROCM(M%) to print out message M%, and then, depending on the value of Z%, jump to the appropriate switched message. Getting at the message is fairly similar to the test method we used while putting the message in the machine:

```
5610DEFPROCM(M%):LOCALI%
5620IFM%=0ENDPROC ELSE P%=m%:I%=1
5630A%=?P%AND&F:B%=?P%DIV16:P%=P%+1
5640IFI%=M%THEN5670
5650FORD%=1TOA%:P%=P%+LEN$P%+1:NEXT
5660P%=P%+2*B%:I%=I%+1:GOTO5630
5670IFA%=0P%=P%+1:GOTO5710ELSEFORD%=1TOA%
5680PRINT $P%:P%=P%+LEN$P%+1
5690NEXT
5710IFB%=0ENDPROC
5720M%=Z%:IFM%>(B%-1)M%=B%-1
5730M%=P%!(2*M%)AND&FFFF:PROCM(M%)
5740ENDPROC
```

Explanations: line 5620 terminates if message 0 is requested, else sets the counter P% to m%, the location in memory where the messages start. It sets the 'current message marker', I%, to 1. Line 5630 reads the number of lines of the current message into A% and the number of switches into B%, then increments P%. The method used is the unpacking-a-nybble we used before. At 5640, we jump to printing if I% equals M%, the required message. Otherwise (5650) we skip through A% lines (which, again, works even if A% is zero), then jumps over 2B% switches, adds one to I%, and returns to 5630. This loop would be neater if we had a REPEAT WHILE structure, since we need to test at the beginning of the loop, not the end, which REPEAT UNTIL is designed for.

At 5670 we start to print the message itself. If it hasn't any lines, skip over the carriage return and try the switches at 5710, else (5670-5690) print out A% lines of text. At 5710, we terminate if there aren't any switches. Otherwise, at 5720, we use M% temporarily to hold which number switch we're jumping to. At 5730 we extract the message number at that switch using Part 5's methods, and call PROCM with that value message.

Notice two things about this procedure. First, it's recursive. By this is meant that the definition of PROCM may use a call to PROCM. Recursion is a very powerful feature of BBC BASIC if used properly. One has to be careful because in theory this could go on for ever: if message 2 switched to message 3, which switched to message 2, etc. It's up to you to ensure it doesn't! Second, A%, B%, and D% are all used. You could make these local if you like; I merely choose not to use them elsewhere, to avoid nasty accidents.

6.3 Other utilities – descriptions, light, etc.

This section will detail the other utilities. The first pair describes a given object – PROCDO(O%) – or room – PROCDR(R%). First is the object description:

```
5220DEFPROCDO(O%)
5230M%=?(FNOL(O%)+4+(FNR(O%)=1))
5240Z%=FNS(O%):PROCM(M%):ENDPROC
```

Line 5230 has to pick out the appropriate message number. This is 3 on from FNOL(O%) if we want the short description, and 4 on for the long description. We choose which by whether FNR(O%) = 1 or not. If it is, the value of the (logical) expression (FNR(O%)=1) is TRUE, or -1 numerically. If not, the value is FALSE, or zero. So line 5230 puts the correct value of the message into M%. All line 5240 has to do is set Z% to O%'s state, and call PROCM(M%).

The room describing procedure has some similar features:

```
5520DEFPROCDR(R%):LOCAL I%,J%
5530IFFNL OR R%=1 ELSEPRINT"It is pitch dark":ENDPROC
5540M%=?(FNRL(R%)+23+FNRP(R%,1))
5550Z%=FNRS(R%):PROCM(M%)
5560J%=TRUE
5570FORI%=1TO15:IFFNR(I%)=R% PROCDO(I%):J%=FALSE
5580NEXT
5590IFJ% AND R%=1 PRINT"Nothing"
5600ENDPROC
```


Line 5530 decides anything can be seen. It uses FNL, which is yet to be written, but returns a logical value TRUE/FALSE depending on whether there is a light of any kind in room R%. So if there isn't a light and we aren't describing room 1 – i.e. the player for whom light is irrelevant – we print 'It is pitch dark' and quit. Once over that hurdle, 5540 decides on short or long description based on room property 1 (visited); the short description is chosen if R% is visited, the long one otherwise. Line 5550 sets Z% to the room's state, and prints the appropriate message. The next line sets a flag J% to TRUE, for use later. Line 5570 describes each of the 15 objects if it's in room R%, and clears J%. Line 5590 is used only for when the room is 1 – i.e. the player. If no objects were mentioned, then we print 'Nothing' to follow after 'You are carrying' or whatever the player's message reads.

A small point: the number of objects has been set to 15 in line 5570, since that's how many there will be in 'ROMAN'. You should of course change the number yourself for other games. The only reason I didn't set the number to a variable at the beginning of the game is because it's so seldom used.

This procedure used FNL, the light function. Here it is:

```
5440DEFFNL:LOCALI%,J%
5450IFFNRP(R,0)THEN=TRUE
5460I%=FALSE:FORJ%=1TO15:IFFNP(J%,0)AND(FNR(J%)=1ORFNR(J%)=R)I%=TRUE:
J%=15
5470NEXT:=I%
```

Line 5450 sets FNL to TRUE if the player's current room (which will have the value R) has property zero (lit). If not, then we cycle through the 15 objects (reset this for other games) to see if any have the property light source and are either carried by the player or in the current room. If one does, finish the loop and return TRUE or FALSE accordingly.

The next pair of procedures relate to the handling of the player's input. The use of direct storage allows us to combine all three of the binary searches we used previously for verbs, things, and special words into a single procedure PROCW(X$,I%). This takes as arguments X$ as a (four-character) string and I% as a flag which marks which vocabulary is to be examined. If I% is 1, check verbs; 2, check things; and

3, check specials. The procedure sets J% to zero if X$ isn't found in the appropriate list, and to its code otherwise. If I% is 1 (i.e. a verb check), K% is set to the verb type, otherwise 1.

```
5750DEFPROCW(X$,I%):J%=0:K%=0:U%=1:ON I% GOTO 5760,5765,5770
5760H%=NC:P%=v%:J%=6:GOTO5780
5765H%=NT:P%=t%:J%=5:GOTO5780
5770H%=NS:P%=s%:J%=5
5780IFX$<FNSTR(U%,I%)ORX$>FNSTR(H%,I%)J%=0:ENDPROC
5790IFX$=FNSTR(U%,I%)M%=U%:PROCSET:ENDPROC
5800IFX$=FNSTR(H%,I%)M%=H%:PROCSET:ENDPROC
5810REPEAT IF(H%-U%)=1J%=0:K%=0:UNTIL TRUE:ENDPROC
5820M%=(U%+H%)DIV2:IFX$=FNSTR(M%,I%) UNTIL TRUE:ENDPROC
5830IFX$>FNSTR(M%,I%)U%=M%:UNTIL FALSE ELSE H%=M%:UNTIL FALSE
5850DEFPROCSET:P%=P%+(M%-1)*J%+4:J%=?P%:IFI%=1THENK%=P%?1ELSEK%=1
5860ENDPROC
```

The structure is very similar to that in Part 3. Line 5750 sets U% to the lower end of the range of vocabulary, which is 1 no matter what part of vocabulary we're checking. Lines 5760-5770 then set H% to the higher end of the range of vocabulary (NC, NT, or NS are the numbers of Commands, Things, and Specials), put our counter P% to v%, t%, or s% – the appropriate location in memory – and set the length of the data entry into J% as 6, 5, or 5 depending on the vocabulary type.

Life now proceeds as before. Line 5780 checks for vocabulary out of range; 5790-5800 for vocabulary at the endpoints of the range. These lines all use FNSTR(M%,I%) which returns the M%th string of type I% (the same flag as in PROCW). We then proceed until we fail to find a match, or, at line 5850, to look up the number of the vocabulary and, if necessary, the verb type. These are then returned in J% and K%.

The function used above, FNSTR, is easily written:

```
5880DEFFNSTR(M%,I%)
5890IFI%=1T%=v%+6*(M%-1)ELSEIFI%=2T%=t%+5*(M%-1)ELSET%=s%+5*(M%-1)
5900!Z=!T%:Z?4=13
5910IFRIGHT$($Z,1)=" "REPEAT$Z=LEFT$($Z,LEN($Z)-1):UNTIL RIGHT$($Z,1)
<>" "
5920=$Z
```

Line 5890 sets T% to point at the first byte of the appropriate vocabulary. Line 5900 dumps the first 4 bytes over to Z – which is a location we haven't defined yet, but will. It will be &C00, normally used for user-defined characters, but not in this game. We shall ensure that this choice doesn't interfere with anything, later. Normally it's bad practice to address

below the normal PAGE value, but it saves quite a bit of room. If you're unhappy, feel free to say DIM Z 40 at the start of your program; it'll achieve the same thing. We then place a 13 (carriage return) after these four characters. Should the string contain blanks (5910) we shrink it. Line 5920 eventually returns the string at location Z, as required.

One other short procedure completes our list. It takes Z$ and converts it to lower case. This will allow the player to type in upper or lower case, provided that all our database is written in lower case:

```
5940DEFPROClc:FORA%=1TOLENZ$:B%=ASC(MID$(Z$,A%,1))OR32
5950Z$=LEFT$(Z$,A%-1)+CHR$B%+RIGHT$(Z$,LENZ$-A%):NEXT:ENDPROC
```

Line 5940 may look a little odd. The point is that lower case ASCII codes are 32 above upper case. Simple adding 32 to the ASCII code of a given character fails for several reasons – if it's already lower case, or a space, or whatever. But logical ORring with 32 will set that particular bit (like a property!) and give the right answer. Line 5950 then replaces that character in Z$ by its lower cased version.

6.4 The overall running program

We may now program in the rest of the 'shell' of the Adventure program, for your use in other Adventures. We begin with the initialisation section 1-99:

```
 8REM ONERRORGOTO1005
 9MODE7:HIMEM=????
10NC=??:NT=??:NS=??:N%=????:Z=&C00
11o%=????:r%=????:v%=????:t%=????:s%=????:m%=????
15Z$=STRING$(20," ")
20FORI%=7TO8:PRINTTAB(10,I%);CHR$(141);"Roman Adventure":NEXT
30PRINT'':PROCM(??)
40PRINT'':PROCINOUT(1):PRINT''
50Q%=0
```

Line 8, at present REMmed out, allows a jump to a 'would you like another game?' line at 1005 in case of ESCAPE being pressed, or some other error. Please do not remove the REM until you are convinced there are no errors in your program! Line 9 sets the screen mode, and then resets HIMEM to a value which at the moment we don't know. I recommend leaving strings like '????' in, because if you forget to set them, the program will fault immediately rather than do something drastic. We'll know where to put HIMEM – which will be one below the beginning of the object

database – when we have written the main program.

Line 10 sets the number of commands, things, specials, and a counter location for the time of day. (I.e. N% is a location in memory, in fact within the objects database, so will be dumped out when the player saves or loads a game. There might be many such odd locations in general. We can access them all through N%?1, etc. These would probably be put just after the objects.) We also set Z, which we've seen before. Line 11 sets the beginnings of the object, room, verbs, things, specials, and message databases. Again we don't yet know what these are! Line 15 initialises the string for player input, Z$. Line 20 gives a banner for the game – Electron users should modify accordingly.

Line 30 prints out a suitable 'story so far' message. Again, we don't know which message it will be. Line 40 lets the player read in the dynamic part of the database, and start the game. Initially this will be on a file we will provide called INIT, but he will certainly make others as he plays, and can start from any of these. This uses PROCINOUT, which will be defined in a moment. Finally, line 50 sets Q%, the player's previous room, to zero, to guarantee a room description first time round.

PROCINOUT(I%) relates to saving and loading the game in its various stages; loading for I% = 1, saving for I% = 2. If you're using a cassette-based system, and wanted to save the game, the file you created could be called anything, since you have to keep track of where on tape each file is. Thus we could require all files to have the same name; INIT, say. But this can't work for disc-based machines because writing a file called INIT onto a disc with a file of that name already present would delete the first file. So we must give the player a choice of names for his file.

Why the problem? Well, we will be using the operating system commands *SAVE and *LOAD, rather than BASIC, to save the database. Unlike SAVE and LOAD, we can't pass BASIC strings to these commands (e.g. we cannot ask the player for a filename in X$ and then say *SAVE X$.) Those with BASIC II have no problem; there's an OSCLI command. Those with BASIC I need the only machine

code call in this book (or use PROCoscli in 'Creative Assembler on the BBC Microcomputer', also available in the Penguin Acorn Computer Library):

```
5960DEFPROCINOUT(I%):PROCM(??):INPUTY$
5970$Z=Y$+" 2A00":IFI%=1$Z="L. "+$Z ELSE$Z="S. "+$Z+" 2D50"
5980X%=0:Y%=&C:CALL &FFF7:ENDPROC
```

Line 5960 prints a specific message, to be defined. It relates to asking about a filename and so will obviously change from game to game. The player supplies its name in Y$. Line 5970 then builds around Y$ either the string '*L. ''Y$'' 2A00' or '*S. ''Y$'' 2A00 2D50'. ('*L.' and '*S.' are shorthand for *LOAD and *SAVE.) Here I've used ''Y$'' to mean whatever the actual string in Y$ was, not the characters ''Y$''! So if Y$ was ''FRED'', we'd have a string '*L. FRED 2A00', for example.

The use of *LOAD and *SAVE is covered in your user guide. They allow the dumping or retrieval of specified sections of memory. Let's look at *LOAD first. Its format is:

*LOAD filename address

which takes the contents of the file whose name is 'filename', and places it into memory beginning at the hexadecimal location 'address'. There! I told you we'd need hexadecimal for something! Now this is an operating system call, so the machine doesn't care in any way what's in 'filename'. It can be BASIC, machine code, word-processed material, databases, or just plain junk! What you do with it is up to you, too. *SAVE is a little more lengthy. Its format is:

*SAVE filename startaddress finishaddress+1

which takes the memory between startaddress and finishaddress inclusive and places it into a file called 'filename'. Again, both addresses must be in hexadecimal, without the '&'; the contents can be anything you please. In our case they're going to be the dynamic part of the database, i.e. from the beginning of the objects (o%) to the end of the rooms (one less than v%).

So, symbolically, we could write '*SAVE filename ''o%'' ''v%''' to save the dynamic part of the database, where we'd insert the values of o% and v% once we knew what they were. Above, I've used the fact that I know they're &2A00 and &2D50 respectively.

That concludes all you need to know about the use of *LOAD and *SAVE. The problem for us is to get the input filename into the operating system. That's where lines 5970 and 5980 come in. We construct the string containing what we want the operating system to do at byte Z (&C00, you'll recall). Line 5980 then does what the manual tells you to do, and the result is passed to the operating system. I don't know how it works either; I just follow instructions and you should do the same! (Incidentally, always get into the habit of writing for the lowest common denominator; don't say 'Ah, but I have BASIC II so I can use OSCLI' because one of your users may not have it!)

Next comes the between-turns section (lines 100-199):

```
100REPEAT R=FNR(0):IFR<>Q% ORNOTFNRP(R,1)PROCDR(R)
110PROCRSP(R,1,1):Q%=R:F%=0
```

We shall store the player's room in object zero's room, as well as letting it be the variable R. The point is that when we do a save or a load, the program must be able to find out where the player is! If the player has moved, or if room R is unvisited, we describe the room. (This makes the response to 'LOOK' trivial: we just unset visited on the current room, and return.) Line 110 sets room R to be visited willy-nilly, and sets Q% to R as well; finally our flag F% is cleared.

Any other 'special' bits you want to add for a given game would go here.

Next comes the player input, in lines 200-299:

```
  200REPEAT
  210IF F%=0 REPEATINPUT'":"Z$:UNTILZ$<>""
  220PROClc
  230J%=INSTR(Z$," "):IFJ%=0X$=LEFT$(Z$,4):Y$="":GOTO300
  240X$=LEFT$(LEFT$(Z$,J%-1),4):Y$=RIGHT$(Z$,LEN(Z$)-J%)
  250IFLEFT$(Y$,1)="  "REPEAT Y$=RIGHT$(Y$,LEN(Y$)-1):UNTIL
LEFT$(Y$,1)<>" "
  260Y$=LEFT$(Y$,4)
```

We prompt him with a colon, and input to Z$, repeating if he merely hits carriage return. Line 220 lower-cases Z$. We then follow exactly the same sequence as in 'MINI' to split Z$ into up to two words, both of at most 4 characters.

Next comes the vocabulary handling, in lines 300-399:

```
  300PROCW(X$,1):C%=J%:IFJ%>0THEN330
```



```
  310Y$=X$:PROCW(Y$,2):D%=J%:PROCW(Y$,3):IFD%+J%=0PRINT"EH?":UNTIL FALSE
  320PRINT"What do you want to do with the ";Y$;"?":INPUTZ$:PROClc:
X$=LEFT$(Z$,4):PROCW(X$,1):C%=J%:IFJ%=0PRINT"EH?":UNTIL FALSE
  330D%=K%:IFD%=0ANDY$<>""PROCM(??):UNTIL FALSE
  340O%=0:S%=0:IFY$=""THEN360
  350PROCW(Y$,2):O%=J%:IFJ%>0ELSEPROCW(Y$,3):S%=J%:
IFJ%=0ANDD%<>3PROCM(??):UNTIL FALSE
  360IFD%=1ANDS%>0PROCM(??):UNTIL FALSE
  370IFD%>0ANDD%<3ANDY$=""PRINTX$;" what?":INPUTZ$:PROClc:
Y$=LEFT$(Z$,4):GOTO330
  380UNTIL TRUE
```

Line 300 checks X$ – the presumed command – against its vocabulary using PROCW, and puts the command number into C%. As long as this is positive, we can proceed to 330 to check for second words. (I apologise for the GOTOs here, but the lines are too long for IF/THEN constructs.) If not, it wasn't recognised (line 310). Then we try it out as a second word (Y$ = X$) and test this hypothesis for things and specials. If it wasn't either, we print 'EH?' and get some more input. If we did recognise the first word as a thing or a special (i.e. a proper second word) we ask what the player wants to do with it; lower-case the result and truncate it to 4 characters, and see if that is understood as a first word. If not, we give it up as a bad job!

By line 330, we have an understood first word, so we store its type in D%. If there's a second word and there shouldn't be, we print a message of as yet unknown number saying 'I don't understand that!' and quit. Notice that we use D% in PROCM and to hold the verb type. However, no messages get printed if all is well, so the overlap doesn't matter. At line 340, we're probably expecting a second word, so we set O% and S% to zero, and skip to line 360 if there was no second word. Line 350 has to check that second word. We try it as a thing, and then as a special word. Only if the verb type is 3 (a new numbering system to be detailed later) which allows unrecognised second words) can we proceed, else we didn't understand. Line 370 finds that D% = 1 or 2, which will require a second word. If there wasn't one, ask for it, lower-case it, and skip back to check it out at line 330. If all was well, end the REPEAT loop at 380.

Next will come the preprogram at lines 400-499. This will be totally specific to the individual game and so can't be written here.

The command handling routine comes at lines 500-599:

```
500F%=0
510ON C% GOSUB ????,????,... etc.
520IFF%=9 PROCDIE
530IFF%=1Z$=Y$:GOTO200
540IFF%=2PROCNEWGAME
```

which is again a little sketchy because the important line is game-dependent. We set F% to zero at line 500, and then GOSUB at 510 to whichever chunk of program C% pointed at. At 520 we see if it was fatal (F%=9); if so, to the deathprog at line 1000. Line 530 checks if the command required ignoring the first word, and treating the second as a new first word; if so, make the changes and back to 200. Finally, 540 checks to see if the player wants to stop (F%=2); if so, let him. Nearly all of this is identical with 'MINI'.

This leaves only death and restarting, at lines 1000-1005:

```
1000DEFPROCDIE:PROCM(??):GOSUB2730:PROCNEWGAME
1005DEFPROCNEWGAME:PROCM(??):INPUTX$:IF(ASC(X$)OR32)<>110RUN ELSEEND
```

Line 1000 delivers a suitable death message, then calls the scoring subprogram (more of which later). In a general case, this line number would vary, of course. I just didn't want to leave in too many question marks! Line 1005 asks if the player would like another game, then scans the first character of his response, lower-cased, to see if it's 'n'. If it is, end the program, else re-run.

At lines 6000 and beyond we shall tuck in the exit programs, which are stored, remember, with the exits they belong to.

And that completes the 'shell' program. We may now proceed to use this for the game 'ROMAN' (at last!)

7 PROGRAMMING AN ADVANCED ADVENTURE GAME: 'ROMAN'

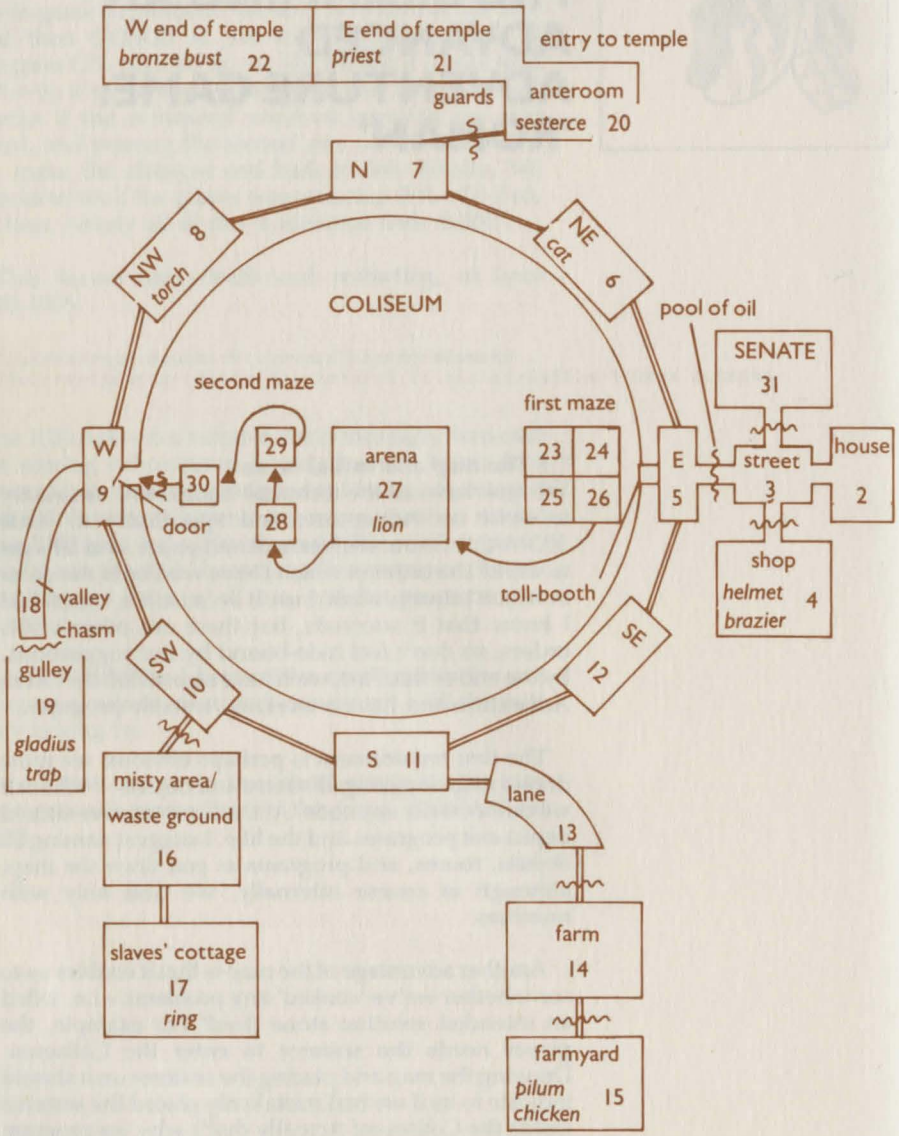
7.1 The map and initial layout

We now have all the technical 'equipment' necessary to write an Adventure, and specifically to write 'ROMAN'. So out with pencils and paper, and let's get to work! The order in which I have worked is the order in which I always work. I use it because it is logical and I know that it succeeds, but there are other viable orders, so don't feel hide-bound by my suggestions. By the end of this Part, we'll have assembled the entire Adventure and have a working, testable program.

The first requirement is perhaps obvious: we must draw a map depicting all rooms and objects. Without it we cannot write any code. At the same time we should depict exit programs and the like. I suggest naming all objects, rooms, and programs as you draw the map, although of course internally, we deal only with numbers.

Another advantage of the map is that it enables us to see whether we've 'cooked' any problems – i.e. killed an intended solution stone dead! For example, the player needs the sesterce to enter the Coliseum. Drawing the map and placing the sesterce on it should indicate to us if we had mistakenly placed the sesterce inside the Coliseum! Actually that's why we program

in pencil – because there will be mistakes. . . Here is my version of the map. It's about Mark 3, in fact; typically there are one or two rough versions as I get my ideas straight. The act of drawing often suggests a puzzle or two!



I've indicated with various symbols where exit programs occur, together with most of the exits. In the case of mazes, these get fairly complicated, as does showing multi-level maps; in which case, do the best you can! You'll notice that no exit patterns have been shown for the two Coliseum mazes; we'll fill those in later.

You should also decide on an 'official route' for the game and make sure, by reference to the map, that this can be followed successfully. Here's my order:

1. Get helmet (in and out of shop)
2. Pass through oil once
3. Do valley, get gladius and throw it; collect mousetrap
4. Leave trap at farm, get pilum and pick up mouse
5. Get cat (losing mouse); get chicken (losing cat)
6. Wear helmet to pass guards; drop helmet and get sesterce
7. Into temple, kill chicken (losing gladius to priest)
8. Get bust, leave temple and re-wear helmet
9. Past guards again
10. Get wood
11. To shop through oil again
12. Light wood, catch fire
13. To misty area, dropping wood just before entering it
14. Be duly put out, then go back to wood – which is why it must be dropped there, as else can't move from dark to dark
15. Collect ring from cottage
16. Pay at toll-booth
17. Through maze 1 into arena
18. Throw pilum to kill lion (losing pilum)
19. Collect wreath and NEW hint
20. Enter maze 2, then go N, E, W to leave Coliseum (memo – need a message when we leave!)
21. To senate
22. Throw torch when confronted by senator, to win game.

Do check the route listing against the map when you get to this stage. It can reveal fundamental flaws in game-planning at a stage when they can be cured with the stroke of a pencil.

It's worth noticing the odd touches. We avoid people trying 'PAY SHOPKEEPER' for the helmet

because the player can't get the sesterce until after he's got the helmet – which means there's no real need to pay anything then! Mind you, he might still say it, in which case an uninformative 'You can't do that!' will handle him. The brazier had better be untakeable, as had various other objects like priest, lion, etc. – otherwise we'll have an embarrassing situation on our hands. Notice also that both exits from the farmhouse must have the 'mouse-producing' program set on them – we can't be sure how the player will tackle that problem, after all.

During the rest of the coding, your map should always be in front of you, as you'll refer to it constantly.

7.2 The object list

Out with another sheet of paper ready for the next item on our agenda, namely the preparation of the list of objects. First what properties do we need (remember we have up to eight available)? Already the database handler assumes property 0 is LIGHT SOURCE, and we will keep that as it is. The brazier and some other objects will be untakeable, and will be assigned property NOTAKE. The property TREASURE will simplify the scoring. Property KILLABLE will also ease the handling of KILL; EATABLE will do the same for EAT. Finally, as we're using states for the wood, we might as well define a property OILY, although we do not really need it. Thus our list of properties, together with the numbers I've given them, are:

```
0   LIGHT SOURCE
1   NOTAKE
2   TREASURE
3   KILLABLE
4   EATABLE
5   OILY
```

Write this list down on top of the 'object' sheet of paper so that you don't lose it. A quick count of objects in the game yields 15. Notice that that the senator doesn't make it into the list – he's a set of programs and descriptions. You could quite easily make him into an object if you prefer.

Now we make a table, with headings like this:


```
NUMBER OBJECT STATE PROPERTIES ROOM SHORT LONG
```

and proceed to fill it. The order of the objects is of course arbitrary. I strongly recommend filling in both the number of the object and your name for it, even though that doesn't get into the database. The point is that when you want to refer to the table to see what number the pilum was, you're thinking 'pilum', not '9'! So provide both.

Well, here's my list:

NUMBER	OBJECT	STATE	PROPERTIES	ROOM	SHORT	LONG
1	brazier	0	0,1	4	0	1
2	helmet	0	-	4	2	4
3	cat	0	3,4	6	5	6
4	torch	0	-	8	7	11
5	gladius	0	-	19	15	16
6	trap	0	-	19	17	18
7	ring	0	2	17	19	20
8	chicken	0	3,4	15	21	24
9	pilum	0	-	15	27	28
10	sesterce	0	0	20	29	30
11	priest	0	1,3	21	0	31
12	bust	0	2	22	32	33
13	lion	0	1,3	27	0	34
14	wreath	0	2	0	35	36
15	mouse	0	4	0	37	38

Let's go through these just to check you understand everything. All the states are initially zero; for other games this might not be the case, of course. The brazier has properties LIGHT SOURCE and NOTAKE; the first because that's logical – it's full of burning coals – and the second because we don't want it taken. Thus it has no short message associated with it, because the player will never hold it. It has a long message, however. Out with another sheet of paper – which will stretch to several – dedicated to writing out messages. This is number 1, then:

1
There is a brazier of glowing coals here.

The helmet has no properties at all, but has both short and long messages. The short message will vary depending on the state of the helmet (state 0 means not worn, state 1 means worn). The long message won't vary, because if the helmet's on the ground it can't be being worn, can it? So the short message contains a switch:

2
(switches 0, 3) A helmet

3
(which you are wearing)

4
A military helmet lies nearby.

Just check to make sure you understand the switching. If the helmet is in state zero, message 2 prints 'A helmet'. If it's in state 1, the player will see

A helmet
(which you are wearing)

The cat has properties 3 and 4: KILLABLE and EATABLE.

5
A contented cat

6
A tabby cat frisks here.

It needs only simple messages because we won't actually let the player kill it (which would necessitate more messages, as we'll see below when we describe that poor chicken). Nonetheless, we don't want the KILL program to retort 'You can't do that!' to 'KILL CAT', which it will do if we fail to set KILLABLE on the cat.

The torch is a little more complicated. Its state will vary dramatically during the game. Initially it has no properties, but it will acquire several. Its messages are 7 and 11, each of which is switched:

7
(switches 8, 9, 10) -- null --

8
An unlit torch

9
A burning torch

10
A blackened stump

11
(switches 12, 13, 14) -- null --

12
A tapered piece of wood as long as your arm lies here.

13
There is a burning torch here.

14
There is a black stump here.

These reflect the torch in state 0 (unlit), state 1 (burning merrily) and state 2 (useless and burned up). The gladius is straightforward:

15
A gladius

16
A vicious-looking gladius lies here.

So is the mousetrap:

17
A trap

18
There is a contraption of sharp iron, wood and cheese here.

Notice, please, the hints in messages 16 and 18 as to the objects' uses. Being kind to the player is a useful habit to acquire! The ring, being treasure, has property 2 set, and just two messages:

19
A ring

20
There is a silver ring, stolen by the slave, here!

The standard Adventure 'signal' for treasure – an exclamation mark – is visible here, though if the player doesn't realise that's what he's after, he shouldn't be playing games like this. Next comes the chicken, which, like the cat, has properties 3 and 4. Because it can be alive (state 0) or dead (state 1) it needs message switching:

21
(switches 22, 23) -- null --

22
A chicken

23
A dead chicken

24
(switches 25, 26) -- null --

25
A chicken struts around, clucking.

26
A dead chicken lies sadly here.

The pilum and sesterce are simple enough, needing one pair of messages apiece:

27
A pilum

28
Someone has left a pilum here.

29
A sesterce

30
An old sesterce is on the floor.

Note the variant type of description in message 28; it just makes a change! The priest, object 11, is NOTAKE but also KILLABLE. This may sound odd, but the cat is the same. We'll actually kill the player if he tries to kill the priest! Since he's untakeable, the priest only gets a long message:

31
A priest, his hands red with blood, looks at you expectantly.

This message again contains a hint about sacrificing things. Otherwise I claim it's a very long haul for the player's mind to guess the correct action here – and recall that he doesn't get any second chances, either. The bust has property 2:

32
A bronze bust

33
A bronze bust of Cicero is yours for the taking!

Again, we signal treasure to the player. The lion, object 13, has NOTAKE and KILLABLE (in this case, he really is!) and thus only one message. We may as well use it to clue the player into the situation and do away with a second message. Instead of 'There is a lion here' plus 'As you enter, he bears down on you', we have:

34
A roaring lion bears down on you, its jaws agape!

The final two objects begin life in the destroyed room, room zero. They are the wreath, to be given to the player on killing the lion, and the mouse, to be delivered to the farmhouse. The wreath is treasure, and the mouse is – ugh! – EATABLE:

35
A gold wreath

36
The gold wreath of victory is here!

37
A dead mouse

38
There is a dead mouse here.

That finishes the objects. Fifteen objects have generated 38 messages! You may begin to understand my earlier comments about computer space. Good space handling distinguishes the good programs from the mediocre.

7.3 The room list

Set the object list to one side for future reference, and let's move on to the room list; there are 31 rooms, you'll remember, including zero (the destroyed room) and one (the player's carried objects).

The table is in the same form as the one we drew up for objects. But the 'room of object' column is replaced by an 'exits and exitprogs' column, which should be fairly wide because maze rooms, for example, tend to have many exits!

What room properties shall we define? Again, property zero is already defined (LIT), as is property 1 (VISITED). Only two other useful ones come to mind: NODROP, for use in the mazes, and JUMPSPEC, for use on either side of the chasm, to indicate that JUMP is treated specially there. So our final list of properties reads:

```
0    LIT
1    VISITED
2    NODROP
3    JUMPSPEC
```

Jot this list down on top of the first page of the room list for future consultation.

Because there are so many exits, I'll detail each room on the list as we go along. We begin with room 1, the player:

```
NUMBER NAME    STATE   PROPERTIES EXITS  SHORT LONG
1      player   0         0       none    0   39
```

The 'player room' is lit in case we later fool around with the room description routine ('It is pitch dark' would be a stupid response to INV!) Since room 1 is never VISITED, we'll always need the long description:

39
You are carrying:

The first 'real' room is 2:

```
2      house     0         0       W 3    40   41
```

As with all rooms, its original property is zero (lit). It has a single exit West, to room 3. Its messages are:

40
You're at home.

41
You're in your house, which is poor but comfortable. To the west lies a street.

Room 3 is the street, a little more complicated. If in doubt, check the map here:

```
3      street    2         0       E 2    42   43
                                   N 31 P1 (senate
                                   entrance)
                                   S 4
                                   W 5 P2 (oil
                                   pool)
```


We have to keep track of how much the oil pool is used, and I've chosen to track this by the state of the street. This is set at 2 originally, so the player can make two passes through the oil – he'll need them both, as you'll see by looking at the map. Once through to get the torch, and once back to light it. The exits to East and South are ordinary. That to the North needs exit program 1, the senate entrance – I'm using 'P1' as shorthand for 'exit program 1' in the room list. That program will attend to getting stabbed, or confronting the senator. To the West, we need program 2, the oil pool program. We'll use it again in a minute! The appropriate messages are:

42
You're in the street.

43 You are in a long east-west street. To the north lies the Senate, and to the south is a small shop.

On to the shop, room 4:

```
4      shop      1         0       N 3 P3 (shop-  44   45
                                   keeper)
```

The shop begins life in state 1 – i.e. with a shopkeeper in the room description. After leaving North, and executing program 3, we'll lose the shopkeeper. Here are the messages, which contain a switch in the long description but, cunningly, not in the short:

44
You're in the shop.

45
(switches 0, 46) You are in an old shop, with its exit northwards.

46
A shopkeeper is keeping his eye on you.

Thus the shopkeeper is only mentioned in state 1! Now we begin the long round of Coliseum rooms. Most of these have identical long and short descriptions. This obviously saves space, but also helps the player not make a mistake when he's running furiously round to put his fire out.

```
5      E Colis   0         0       W 5 P4         47   47
                                   (entrance to
                                   Colis.)
                                   NW 6
                                   SW 12
                                   E P2 (oil pool)
```

Let's look at the exits. West actually takes the player to the room he's already in, room 5. But the room description will refer to interesting things to the West, and the player needs some response. To the East lies the oil pool program which will work from either direction. The long and short messages are the same, namely:

47
You're east of the Coliseum, a large circular building. A road around it leads northwest and southwest. There is a toll-booth west and a street east.

Room 6 is straightforward:

```
6       NE Colis  0          0     W 7    48    48
                                    SE 5
```

with again a single message:

48
You're northeast of the Coliseum. The road goes west and southeast.

Room 7 will get used a fair amount, so we'll give that a short message as well:

```
7       N Colis   0          0     W 8    49    50
                                    E 6
                                    NE 20 P5
                                    (guardprog)
```

49
You're north of the Coliseum.

50
You're north of the Coliseum. The road goes east and west, and an archway leads northeast through some barracks.

Rooms 8 and 9 are normal:

```
8       NW Colis  0          0     E 7    51    51
                                    SW 9

9       W Colis   0          0     NE 8   52    52
                                    SE 10
                                    W 18
```

51
You're northwest of the Coliseum; the road goes east and southwest.

52
You're west of the Coliseum; the road goes northeast

and southeast. A valley stretches west, and a closed door bars the way east.

Room 10 will change its description after the misty area has been solved, so its message structure is more complicated. We also need an exit program for the SW exit:

```
10      SW Colis  0          0     NW 9   53    53
                                    E 11
                                    SW 16 P6 (mist)
```

53
(switches 54, 55) You're southwest of the Coliseum; the road goes northwest and east.

54
To the southwest is a dank, misty area.

55
To the southwest is some waste ground.

Rooms 11 to 13 are normal:

```
11      S. Colis  0          0     W 10   56    56
                                    E 12
                                    SE 13

12      SE.Colis  0          0     NE 5   57    57
                                    W 11

13      Lane      0          0     NW 11  58    58
                                    S 14
```

56
You're south of the Coliseum; the road leads east and west. A narrow lane goes southeast.

57
You're southeast of the Coliseum; the road leads northeast and west.

58
You're in a lane winding from northwest to south.

```
14      Farm      0          0     N 13 P7 (mouse)  59    60
                                    S 15 P7
```

and messages:

59
You're in the farm.

60
This is a crude farm with little furniture. Small holes dot the base of many of the walls. Doors lead north and south.

Rooms 15 to 17 are normal. Note, though, that room 16 is never described as dank and misty; the only time the player gets in there is when it isn't so any more!

```
15      Farmyard  0          0     N 14     61    61

16      Waste gr. 0          0     NE 10    62    63
                                   S 17

17      Cottage   0          0     N 16     64    64
```

We save on messages here as the player is unlikely to return to places like the cottage – thus no short description is needed:

61
You're in an enclosed farmyard. The only exit is north.

62
You're on the waste ground.

63
You are on some damp waste ground. A cottage is south, and the road is to the northeast.

64
You find yourself in a cottage, the hideout of the slave. The only exit is north.

The valley and gully are unusual because they have property 3 (JUMPSPEC), and because the gully has no exits!

```
18      Valley    0          0,3   E 9      65    66

19      Gully     0          0,3   none     67    67
```

with messages:

65
You're in the valley.

66
You are in a valley curving from the east and ending at a chasm to the south. A gully is visible across the chasm.

67
You're in a gully across the chasm, with no obvious exits.

The anteroom has programs on all exits. Since its

description will refer to stairs up to the northwest, we arrange for exits up and northwest, both going to the east end of the temple via the priestprog. We can re-use the guardsprog on the other exit:

```
20      Anteroom  0          0     NW 21 P8 68    69
                                   (priest)
                                   U 21 P8
                                   SW 7 P5
```

Messages:

68
You're in the anteroom.

69
You find yourself in an anteroom to the temple, to which steps lead up to the northwest. A passage leads back southwest through the barracks.

Rooms 21 and 22 are normal. The west end of the temple only gets a long message:

```
21      E. temple 0          0     W 22     70    71
                                   SE 20
                                   D 20
```

Messages:

70
You're at the east end of the temple.

71
You are at the eastern end of an east-west temple to Zeus. Stairs exit down to the southeast.

72
You're at the west end of the temple.

```
22      W. temple 0          0     E 21     72    72
```

Now come rooms 23 to 26, the first Coliseum maze. These have property 2 (DROPLOSE) and have to satisfy certain requirements. First, each must have a unique exit that the others don't have, so the player can map the maze. Room 23 has an E exit, room 24 a N exit, room 25 a S exit, and room 26 a W exit. Second, room 26 must be hard to find: so only room 25 goes there. Indeed, there's a tendency to move back to 23.

Notice also that an exit from a room may lead back to

the room the player's already in: E from room 23 will lead to 23, for example. And it's vital that we place an exit program here! If not, the program will duly move the player to the room he's already in, but fail to print out any room description because to the program, the room is still VISITED. Thus all the player will see is the colon prompt, which breaks the cardinal rule that all player actions must get some sort of response.

To prevent this, we add repeatprog to all such exits, which merely unsets VISITED on the current room, thus ensuring its description next time round. We must also, by the way, ensure that long and short descriptions are the same in a maze (or alternatively set all rooms VISITED beforehand) otherwise a long description will indicate to the player that he has reached a new room. We don't want to be too helpful, do we?

```
23      1st maze1 0          0,2    E 23 P9    73    73
                                    (repeatprog)
                                    NE 24
                                    NW 24
                                    SW 24

24      1st maze2 0          0,2    NW 23    73    73
                                    N 25
                                    SE 23
                                    NE 24 P9
                                    (repeatprog)

25      1st maze3 0          0,2    SW 26    73    73
                                    S 23
                                    SE 25 P9
                                    NW 24

26      1st maze4 0          0,2    W 27     73    73
                                    SW 23
                                    SE 24
                                    NE 25
```

The sole message reads:

73
You're in a maze of milling crowds, here for the Games, and jostling you about.

The arena is normal, with but a single exit to the second maze:

```
27      Arena      0          0     SW 28    74    74
```

and message (only one is needed, the player won't be coming back):

74

You are in the Coliseum arena, surrounded by excited crowds. A postern gate leads southwest out of the arena.

The next three rooms are the second maze. We can use the same room description. Note the exit pattern, with all exits but the correct one leading back to room 28. Room 28 itself needs the repeatprog set. Again, property DROPLOSE is set. The final exit, to west of the Coliseum, deserves a message to congratulate the player, so we set gateprog on the exit.

```
28      2nd maze1 0          0,2    N 29     73    73
                                    NE 28 P9
                                    E 28 P9
                                    SE 28 P9
                                    S 28 P9
                                    SW 28 P9
                                    W 28 P9
                                    NW 28 P9

29      2nd maze2 0          0,2    E 30     73    73
                                    N 28
                                    NE 28
                                    SE 28
                                    S 28
                                    SW 28
                                    W 28
                                    NW 28

30      2nd maze3 0          0,2    W 9 P10    73    73
                                    (gateprog)
                                    N 28
                                    NE 28
                                    E 28
                                    SE 28
                                    S 28
                                    SW 28
                                    NW 28
```

There are no new messages of course. The final room is the Senate itself. Since we aren't setting the senator as an object, we'll handle all the final action with room states (you don't have to do this – I'm merely demonstrating the flexibility of the database system). The senate's original state is 2; every beat the player spends in it, reduces it by one in postprog. When it hits zero, the player dies by stabbing. Thus he gets his turn of entry (down to state 1) and then a turn to defeat the senator, as we desire. Thus we get:

```
31      Senate     2          0     none     75    75
```

and message:

75
You are in the Senate, a luxurious area, but unlit at this

late hour. Ganopus meets you by the bathing pool. Instead of taking your three treasures, he treacherously draws a knife and moves towards you to silence you forever!

This has built in the senator – now named Ganopus (which will now have to be part of the vocabulary).

That concludes the room list. For the sake of clarity, here's the complete list:

NUMBER	NAME	STATE	PROPERTIES	EXITS	SHORT	LONG
1	player	0	0	none	0	39
2	house	0	0	W 3	40	41
3	street	2	0	E 2 N 31 P1 (senate entrance) S 4 W 5 P2 (oil pool)	42	43
4	shop	1	0	N 3 P3 (shop-keeper)	44	45
5	E Colis	0	0	W 5 P4 (entrance to Colis.) NW 6 SW 12 E P2 (oil pool)	47	47
6	NE Colis	0	0	W 7 SE 5	48	48
7	N Colis	0	0	W 8 E 6 NE 20 P5 (guardprog)	49	50
8	NW Colis	0	0	E 7 SW 9	51	51
9	W Colis	0	0	NE 8 SE 10 W 18	52	52
10	SW Colis	0	0	NW 9 E 11 SW 16 P6 (mist)	53	53
11	S. Colis	0	0	W 10 E 12 SE 13	56	56
12	SE.Colis	0	0	NE 5 W 11	57	57
13	Lane	0	0	NW 11 S 14	58	58

14	Farm	0	0	N 13 P7 (mouse) S 15 P7	59	60
15	Farmyard	0	0	N 14	61	61
16	Waste gr.	0	0	NE 10 S 17	62	63
17	Cottage	0	0	N 16	64	64
18	Valley	0	0,3	E 9	65	66
19	Gully	0	0,3	none	67	67
20	Anteroom	0	0	NW 21 P8 (priest) U 21 P8 SW 7 P5	68	69
21	E. temple	0	0	W 22 SE 20 D 20	70	71
22	W. temple	0	0	E 21	72	72
23	1st maze1	0	0,2	E 23 P9 (repeatprog) NE 24 NW 24 SW 24	73	73
24	1st maze2	0	0,2	NW 23 N 25 SE 23 NE 24 P9 (repeatprog)	73	73
25	1st maze3	0	0,2	SW 26 S 23 SE 25 P9 NW 24	73	73
26	1st maze4	0	0,2	W 27 SW 23 SE 24 NE 25	73	73
28	2nd maze1	0	0,2	N 29 NE 28 P9 E 28 P9 SE 28 P9 S 28 P9 SW 28 P9 W 28 P9 NW 28 P9	73	73
29	2nd maze2	0	0,2	E 30 N 28 NE 28 SE 28 S 28 SW 28 W 28 NW 28	73	73

```
30      2nd maze3 0          0,2    W 9 P10    73    73
                                    (gateprog)
                                    N 28
                                    NE 28
                                    E 28
                                    SE 28
                                    S 28
                                    SW 28
                                    NW 28

31      Senate     2          0     none    75    75
```

7.4 The exit programs

Keep the room list by you for easy reference too. The next part of the programming actually involves writing some BASIC! We do the exit programs next, largely because they should still be fresh in your mind after writing the room list. Don't put these on the machine yet. Program on paper, with plenty of verbal comments by the lines of program. In a week's time you'll need to understand what you meant by line 6050. A pencilled comment alongside is just as efficient as a REM in BASIC, but takes up no space!

Each exitprog will be discussed in turn. They may use three of the work variables, I%, J%, and K%; L% will be storing something in the main moving command and should be left alone. The first is the senate entrance, P1:

```
6000IFFNR(7)<>10RFNR(12)<>10RFNR(14)<>1PROCM(76):F%=9:RETURN ELSE
PROCM(77):RETURN
```

The idea is not to let the player enter unless he has all three treasures; so we check that objects 7, 12, and 14 are all held. If not, we assassinate the player with message 76, set the fatal flag (F% = 9) and return. If he has all the three treasures, we let him in, and say via message 77 that the assassins have all gone home:

76
As you enter, a group of senators leap on you, mistaking you for Caesar. They plunge their daggers into you as one man, before noticing their sad error.

77
The conspirators have given up waiting for Caesar and gone home.

The next program, P2, is the oil pool:

```
6020I%=FNRS(3):IFI%=0RETURN
6030PROCM(77+I%):IFFNR(4)=1PROCOSP(4,5,1)
6040PROCRSS(3,I%-1):RETURN
```


We first check (6020) to see if the pool has run out – i.e. examine the state of room 3. If so, ignore the exitprog. Otherwise (6030) send message 78 or 79, and if object 4, the torch, is being carried, make it oily (property 5). Finally, decrease the state of room 3. The messages are:

78
(switch 80) You stride through a half-full patch of oil.

79
(switch 80) You stride through a patch of oil.

80
You and your belongings are soaked.

which both use message 80, to save storage. This trick is very useful, especially as the size of your Adventures gets bigger.

The shopkeeper program, P3, is trivial:

```
6060PROCRSS(R,0):RETURN
```

as it always sets the state of the shop to zero. To be sure, we only ever need do it once, but it doesn't hurt to make it automatic.

The entrance to the arena, P4, merely involves an appropriate message depending on time of day:

```
6090PROCM(82+FNRP(R,0)):RETURN
```

FNRP(R,0) will be -1 (TRUE) if it's daylight, and 0 (FALSE) if it's after dark. So we print message 81 in the light, and 82 in the dark:

81
The booth is closed, as the games don't start till dark.

82
The man on the booth demands payment and won't let you in otherwise.

which also delivers a fairly hefty hint about the verb 'PAY'!

The guards program, P5, takes various forms depending on what the player has done with the helmet:

```
6120IFFNR(2)<>1PROCM(83):G%=1:RETURN
6130IFFNS(2)=0PROCM(84):G%=1:RETURN
6140PROCM(86):RETURN
```

If object 2 (the helmet) isn't held, print message 83 and abort the exit. This involves setting another flag, G%, to 1, which simply marks the fact that the player's destination as scheduled in the static database is not his actual destination. In the main moving program – of which, more later – we shall set G% to zero, try to move, and see if G% is 1. If so, we'll forget the exit. By line 6130, the player has the helmet. If he isn't wearing it, out with message 84 and abort the exit. Otherwise (6140), tell him how clever he is, and let him pass. The messages are, using 85 for both messages 83 and 84:

83
(switch 85) The guards see you are not a soldier.

84
(switch 85) The guards see your helmet, but realise you are not a soldier.

85
They bar your way.

86
The guards assume you are a soldier, and let you pass.

The next program, P6, is the exit to the misty area. We'll have to decide now how we're going to handle the burning player. I've chosen to set the state of the player to a nonzero value while he's burning; in fact it will count downwards in postprog until the player burns up. We don't yet have to work out exactly how many game turns are involved in that, simply note that that's what we are going to do. Thus we have:

```
6170IFFNRS(R)RETURN
6180IFFNRS(1)=0F%=9:PROCM(87):RETURN
6190PROCM(88):PROCRSS(1,0):PROCRSS(R,1)
6200IFFNR(4)=1ANDFNS(4)=1PROCOSS(4,2):PROCOSP(4,0,0):PROCM(89)
6210RETURN
```

Line 6170 asks if the exit has been used once, thus setting the state of the room we're in to 1. If so, ignore the program. Otherwise (6180) if the player's state is zero, he isn't on fire. Hence kill him, and say how we did it (message 87). By 6190 he must have been on fire. We say that the fire has been put out (message 88); turn off the fire; reset the state of the room he's leaving to (a) alter its description and (b) remove the program in future. We now need to check for the torch. If we got to 6200, the player was on fire, and if he has the torch, we put it out (state to 2), deset it as a light source (deset property 0), and tell the player. You may note that the

additional check if FNS(4) = 1 isn't needed (why?). The relevant messages are (note the 'touch' in message 88):

87
In the mist, an escaped slave grabs you and chokes you to death.

88
With a hiss, the mist condenses and extinguishes you. You catch a glimpse of a slave running away.

89
The mist also puts out your torch, worse luck.

Next comes the mouseprog, P7:

```
6230IFR<>FNR(6)ORFNS(15)RETURN
6240PROCR(15,R):PROCOSS(15,1):RETURN
```

If the mousetrap – object 6 – isn't in the room, or the state of the mouse – object 15 – is nonzero, don't do anything. Otherwise (6240) move the mouse to room R, and set its state to 1, thus preventing the program happening again. This program is an example of an 'invisible' one, like the shopkeeper program.

The priest program, P9, merely depends on whether the player has the helmet (he doesn't have to be wearing it):

```
6260IFFNR(2)=1F%=9:PROCM(90):RETURN ELSERETURN
```

Possession of the helmet is therefore fatal:

90
(switch 91) A priest appears. "No soldiers in the temple!" he shouts.

91
You are rapidly arrested and executed.

The reason for the switch is that we can use message 91 again later. I didn't think of this as I was programming the first time, but it is good practice to use messages several times, if you can. That's why we write in pencil!

Program P9, repeatprog, merely unsets VISITED on room R (this works for any of the rooms with P9 set on exits, by the way).

```
6280PROCRSP(R,1,0):RETURN
```

Finally, program P10 is the gateprog, used when the player finally gets out of maze 2 and staggers out to the west of the Coliseum:

```
6300PROCM(92):RETURN
```

which uses:

92
To your relief, you fall out of a door which is slammed behind you.

7.5 The pre- and post-programs

There isn't much to do in the preprogram at line 400. We have to check whether the player's room is lit, in case he moves in the dark. We also must note whether he's in the same room as the lion or priest, so that we can kill him at the end of the turn if he hasn't done the right thing:

```
400Y%=FNL
410FORI%=11TO13STEP2:IFR=FNR(I%)PROCOSS(I%,1)
420NEXT
```

So we set Y% to TRUE/FALSE depending on the lighting, and if either object 11 or 13 is in the player's room, we set its state to 1. In the postprog at line 600 we can see if that's still the case.

Predictably, the postprog is a little more complicated:

```
  600PROCR(0,R)
  610IFY%=0ANDNOTFNL ANDR<>Q%PROCM(93):PROCDIE
  620I%=?N%+1+(?N%=255):?N%=I%:IFI%=35PROCM(94)
  630IFI%=50PROCM(95):FORI%=2TO22:PROCRSP(I%,0,0):NEXT:PROCRSP(31,0,0)
  640I%=FNRS(1):IFI%=0ELSEPROCRSS(1,I%-1):PROCM(96):
IFI%=1PROCM(97):PROCDIE
  650IFFNS(13)PROCM(98):PROCDIE
  660IFFNS(11)PROCM(99):PROCDIE
  670IFR<>31ELSEI%=FNRS(R)-1:PROCRSS(R,I%):IFI%=0PROCM(100):PROCDIE
  680UNTIL FALSE
```

We first move object 0 to room R (that's to allow saving, etc., also to interact with the between-turns part of the main program). Then, at 610, we look to see if the player's moved in the dark: if his room at the start of the turn was dark and his current room was dark and he has moved, kill the player and say so.

Next, at 620, we add one to the time counter stored in ?N%. However, should the player still be going at turn 255, we don't want the counter to get back to zero (as byte storage will) because later on the program will

again say it's getting dark. So we use a logical addition: (?N%=255) is -1 if N%=255, else it's zero. Then, if the player's has 35 turns, we tell him the sun is setting. Why 35? Well, I set it at 180 to start with, and tried out the daylight puzzles. After I felt the player had had long enough, I halted the game and printed ?N%!

Line 630 continues the darkening process; by turn 50 we say the sun has set, and make all the rooms dark except those in the Coliseum – I thought the games might should remain lit!

This concludes the main housekeeping. Now for specific points related to puzzles. Line 640 checks the progress of the burning player. If he isn't on fire, ignore this line. Otherwise, drop his state by one, say he's on fire, and if the state has hit 1, kill the player. Notice the lack of an THEN clause, so that we can have two IFs in the line without chaos.

Lines 650 and 660 check on the lion and priest respectively. If their states are still 1, the player didn't do the right thing. We then kill him accordingly.

Finally, line 670 checks for the endgame puzzle. We drop the state of room 31, the Senate, by 1. If this hits zero, Ganopus gets his man. At 680, finishe the main program loop.

Here are the messages for the postprogram – note the re-use of 91:

93
Thieves kill you while you flounder around in the dark, unfortunately.

94
The sun is setting.

95
The sun has set.

96
You're on fire!

97
You burned to a crisp!

98

The lion grabs you and eats you.

99
(switch 91) The priest shouts "Imposter!"

100
Ganopus' knife slits your throat.

7.6 The vocabulary lists

After that lapse into BASIC, we return to non-programming. Before we can write the main command handlers (which deal directly with what the player says), we need to have the vocabulary sorted out. The system is almost identical to that used in 'MINI', but with a few improvements.

First, the verb type can be one of 4 values:

0 no second word allowed
1 second word must be a thing
2 second word must be a thing or a special
3 any second word may exist, or none

We've come across this already in the 'shell' program.

Next, all directions will be command 1. We shall differentiate between them by their types (and ignore the list above!). In fact, we'll use only types above 3, and the moving program can use the value of the type to find the exit in the database. So North becomes command 1, type 4; NE is command 1, type 6; E is command 1, type 8; and so on, round to D, which is command 1, type 22. By lopping 2 off these types, we get the position of the exit relative to the room label, and will use this later.

The other main improvement is the addition of two extra commands which the player doesn't know about. These are 'magic take' and 'magic move' commands. Their use is purely for debugging, so remove them for the final version. (We do this by putting them at the end of the alphabet; to remove them just reduce NV by two so that the program never looks for them!)

We'll call these commands 'ZZT' and 'ZZM' respectively. 'ZZT object' will take that object even if it's not in the room, unless it can't be taken for some reason! 'ZZM object' will move the player to that object unless it's destroyed.

These two commands allow easy access to bits of the program for testing purposes. Do you want to check out the lion problem? 'ZZT PILUM', followed by 'ZZM LION', will do the trick – you're there in two beats.

All we have to do is make a list, which may as well be in lower case. The first list is for verbs. The numbering is: directions get a 1, all other commands are numbered alphabetically on the order of occurrence of the first synonym. So 'TAKE' has a high position, because 'GET' means the same. The list lives on paper for now, not in the machine.

```
VERBS

catch   inv     pay     south   west
2,2     7,0     5,2     1,12    1,16

d       jump    q       stop    zzm
1,22    8,3     12,0    12,0    19,1

down    kill    quit    sw      zzt
1,22    9,2     12,0    1,14    20,1

drop    light   run     take
3,2     10,1    6,3     2,2

e       look    s       throw
1,8     11,0    1,12    16,1

east    move    sacrifice   u
1,8     6,3     9,2     1,20

eat     n       save    up
4,1     1,4     13,0    1,20

get     ne      say     w
2,2     1,6     14,3    1,16

give    north   score   wait
5,2     1,4     15,0    17,1

go      nw      se      wear
6,3     1,18    1,10    18,1
```

Notice the synonyms. A few useless words like SAY and RUN are also included; their cost is very little, after all. The total is 43 verbs – 41 without the two magic verbs.

Next come the things. Although there are 15 objects, there are 27 things owing to synonyms:

```
THINGS

braz        javelin     stump
1           9           4
```

```
bronze          lion            sword           SPECIALS
12              13              5

bust            money           tabby           booth           slave
12              10              3               1               1

cat             mouse           torch           coliseum        toll
3               15              4               1               2

chicken         pilum           trap            door
8               9               6               1

coin            priest          wood            ganopus
10              11              4               1

contrap         ring            wreath          guard
6               7               14              1

gladius         sesterce                        hole
5               10                              1

gold            silver                          man
14              7                               2

helmet          spear                           oil
2               9                               1

                                                pool
                                                1

                                                shopkeeper
                                                1
```

Finally, we have the special words, 12 in all:

We separate 'man' and 'toll' because we want to allow 'PAY TOLL' or 'PAY MAN' outside the Coliseum.

Having concluded the vocabulary, we can write the main command programs.

7.7 The command programs

We may as well program these in numerical order, although there's no reason why any order is preferable. First comes movement:

1) Movement

```
2000L%=FNRL(R)+D%-2:IF?L%=0PROCM(101):RETURN ELSEJ%=L%?1:G%=0
2010IFJ%>00NJ%GOSUB6000,6020,6060,6090,6120,6170,6230,6260,6280,6300
2020IFF%=90RG%=1RETURN ELSER=?L%:RETURN
```

Line 2000 computes which byte in memory holds the exit corresponding to 'type' D%. If there is no exit, we say so and quit. If there is an exit, we look to see whether there is an exit program involved (J%), and set the abort flag G% to zero. Line 2010 then sends the program on a further GOSUB – we're already in one, from lines 500 to 599 – to the appropriate exit program. We can easily fill in the ten GOSUB addresses because we've already written those in Section 4. On return, at 2020, if the exit was fatal or aborted, we return; else

reset R to the destination. (We could move the player in the fatal case, but he'll never have the chance to discover where he went anyway!) We use one of our work variables, L%, here; so be sure not to use it in any of the exit programs, or chaos will ensue. Obviously one could modify this in 2020 if the need arose. Otherwise, we can use I%, J%, K%, and L% to do all the minor computations in the command sequences. A suitable message is:

101
You can't go in that direction!

2) Get, Take

```
2050IFS%>0THEN3000
2060IFR<>FNR(0%)PROCM(103):RETURN
2070IFFNP(0%,1)PROCM(104):RETURN
2080IFFNR(0%)=1PROCM(105):RETURN
2090IF0%=2AND FNRS(R)ANDR=4PROCM(106):RETURN
2100IF0%<>3ELSEIFFNR(15)=1PROCM(107):PROCR(15,0):PROCR(3,1):RETURN
ELSEPROCM(108):RETURN
2110IF0%=8ANDFNS(8)=0PROCM(109):RETURN
2120PROCR(0%,1)
2130PRINT"OK":RETURN
```

We start by not allowing a special word to be taken. Message 102:

102
You can't do that!

Obviously, this will turn up fairly often, so put its printing at line 3000, and frequently GOTO that line (whose RETURN then acts as a RETURN from the GOSUB, conveniently). Hence we add:

```
3000PROCM(102):RETURN
```

At line 2060, we're talking about an object. If it isn't in the player's room, we say so:

103
That's not here!

and quit. Line 2070 checks if the object is untakeable (property NOTAKE, number 1). If so, we say so and quit:

104
You can't take that!

Continuing on the negative aspects again, just as in 'MINI', perhaps the player is already holding the

object (line 2080)?

105
You're already holding that!

By line 2090, all the necessary qualifications are present to actually take the object. Now, what could stop the player? Nothing fatal (always check fatal problems first). So, let's do each special case. Line 2090 checks whether we're getting the helmet in the shop with the shopkeeper present (can you see why?). If so, we don't let it happen:

106
The shopkeeper won't let you!

Or (line 2100) we can check for the cat (object 3). If we have the mouse, print a message, destroy the mouse, and move the cat to the player; otherwise say he can't pick up the cat:

107
With a loud miaow, the cat jumps into your arms and scoffs the mouse.

108
The cat refuses to be picked up.

Or again (line 2110) it might be the chicken the player's trying to pick up. If it's dead (state 1), no problem; but if state 0, he can't get it:

109
The chicken flutters away from you.

That concludes all the negative thinking; now for positive action. Line 2120 moves the object to the player and (2130) prints "OK". There must always be a response, remember. The 'PRINT "OK":RETURN' line is so useful, we can send other commands there too. Of course, "OK" could have been a message; it just takes up less room as a direct BASIC statement.

3) Drop

```
2170IFS%>0THEN3000
2180IFFNR(0%)<>1PROCM(110):RETURN
2190IFFNRP(R,2)PROCM(111):PROCR(0%,0):RETURN
2200IF0%=2PROCOSS(2,0)
2210IF0%=3ANDR=FNR(8)PROCM(112):PROCR(8,1):PROCR(3,R):RETURN
2220PROCR(0%,R):GOTO2130
```


Explanations: Line 2170 checks for specials again, and won't allow them. Line 2180 objects if the player doesn't have the object he's dropping:

110
You're not holding that!

The object is now droppable. What could happen to avoid that simple action succeeding? In the mazes, (property DROPLOSE, number 2), the object will disappear (line 2190):

111
The object you dropped is soon lost among the feet of the crowd.

If it's the helmet, he certainly isn't wearing it any more, so reset its state to zero (line 2200). Now nothing else unpleasant can happen; what of pleasant things? Line 2210 checks for dropping the cat – object 3 – in the same room as the chicken – object 8. After telling the player, we move the cat to room R and the chicken to the player:

112
On seeing the cat coming, the chicken gives a squawk and flutters up. You grab it and hold onto it.

The only other possibility is that the player actually drops the object without anything happening; hence 2220!

4) Eat

```
2250IFNOTFNP(0%,4)THEN3000
2260IFFNR(0%)<>1PROCM(110):RETURN
2270PROCR(0%,0):PROCM(113):RETURN
```

This is an example of a 'useless' verb. In no sense does eating help the player – but we provide it for 'colour'. If the object isn't eatable, say so (2250). If the player isn't holding it, say so (2260). Otherwise, let him eat it (destroying the object) and say so:

113
It tastes awful, but you force it down your throat.

5) Give, Pay

```
2300IFS%=20%=10
2310IFFNR(0%)<>1PROCM(110):RETURN
```

```
2320IF0%<>10THEN3000
2330IFR<>5ORFNRP(R,0)PROCM(114):RETURN
2340R=23:PROCM(115):PROCR(10,0):RETURN
```

Think negatively as always. The player may have said 'PAY TOLL' or 'PAY MAN' – hence the special words – in which case we convert what he said into 'PAY SESTER' by line 2300. He can't have said 'GIVE MAN' because GIVE has a different code. Line 2310 checks if the player is holding the object. Line 2320 then checks if the object is 10, the sesterce, else refuses to do anything. By 2330, it was the sesterce. If the room isn't east of the Coliseum or it isn't dark, say so; finally, at 2340, everything is fine, so move the player to room 23, tell him, and destroy the sesterce (we don't really want him going through again!)

114
Nobody seems interested in that!

115
You pay your sesterce and enter the crowded Coliseum.

6) Go, Move, Run

```
2360F%=1:IFY$=""PRINTX$;" where?":INPUTY$
2370RETURN
```

This is handled just as in 'MINI', setting the 'second word as first' flag F% = 1.

7) Inv

```
2380PROCDR(1):RETURN
```

This too is the same; just describe the 'player room' and return.

8) Jump

```
2400IFNOTFNRP(R,3)PROCM(116):RETURN
2410IFFNR(5)=10RFNR(9)=1F%=9:Z%=0ELSEZ%=1:R=37-R
2420PROCM(117):RETURN
```

If the room hasn't got JUMPSPEC set, give a silly message and quit. If (2410) the player is carrying the pilum or javelin, kill him; otherwise move him to the room on the other side of the chasm (think about R = 37 − R – it works!). Print message 117, which switches on Z%, set in line 2410:

116
Whoopeee!

117
(switches 118, 119) You leap across the chasm, and scrabble for a handhold.

118
You're carrying something awkward, which causes you to miss your hold and fall to your death.

119
You find a hold and haul yourself to safe ground.

9) Kill, Sacrifice

```
2450IFS%>0ORNOTFNP(0%,3)THEN3000
2460IFR<>FNR(0%)ANDFNR(0%)<>1PROCM(103):RETURN
2470IF0%=11THEN2510ELSEIF0%=13THEN2520ELSEIF0%=8ELSEPROCM(120):RETURN
2480IFFNR(8)<>1PROCM(110):RETURN ELSEIFFNR(5)<>1PROCM(121):RETURN
2490PROCOSS(8,1):PROCOSP(8,3,0):PROCM(122)
2500IFR<>FNR(11)RETURN ELSEPROCOSS(11,0):PROCR(11,0):PROCR(5,0):
PROCM(123):RETURN
2510PROCM(124):F%=9:RETURN
2520IFFNR(9)<>1PROCM(121):RETURN ELSE0%=9:GOTO2800
```

This is among the more complicated subprograms, as there are a variety of killable things around! At 2450 we are negative; if it's a special, like the shopkeeper, or hasn't got KILLABLE set, we don't let the player do it. In 2460 we examine whether the object is either in the player's possession or in his room; if not, object again. The reason for this double check is that it's possible to try killing objects the player isn't carrying, for example the lion. Next, at 2470, we split to various lines depending on what the object being killed is: if the priest (11) to 2510, if the lion (13) to 2520, if the chicken, to 2480. This only leaves the cat, and we won't let the player kill that (message 120).

Line 2480 tries to kill the chicken. Thinking negatively, the player must be holding it – otherwise say so, and quit. Unless he's also holding the gladius, he's wasting his time; we tell him in such a way that what we say will serve for other combinations here (message 121). By 2490 he has satisfied the chicken-killing requirements, so we kill it by changing its state, making it not KILLABLE any more, and delivering message 122. Should the player kill it anywhere other than room 11, under the eyes of the priest, that's the end of that. But in that case (line 2500), we set the priest's state back to zero ('turning him off'); move him to 'destroy' along with the gladius; and print message 123. Some thought will convince you we don't need to check if the priest is in room 11 when we do this (why?).

Line 2510 kills the player for trying to kill the priest. Message 124 uses the ubiquitous switch to 91 again. Line 2520 tries killing the lion. Should the player have no pilum – object 9 – we mutter about no suitable weopon and quit; postprog will take care of the player! Otherwise we reset the object to be 'pilum', and go off to 2800, which is (will be?) the THROW routine. In other words, we redefine the player's action from KILL LION to THROW PILUM, and let THROW worry about what happens. I dislike 'hanging GOTOs', but this one saves rewriting the same code twice.

120
You can't bring yourself to kill the cat, alas.

121
You have no suitable weapon.

122
You slice its head off with the gladius.

123
The priest nods approvingly, removes your gladius with a prayer, and leaves.

124
(switch 91) You attack the priest, who calls loudly for help.

10) Light

```
2550IF0%<>4THEN3000ELSEIFFNR(4)<>1PROCM(110):RETURN
2560Z%=FNS(4):IFZ%PROCM(124+Z%):RETURN
2570IFR<>4PROCM(127):RETURN
2580Z%=-FNP(4,5):PROCM(128):IFZ%=0PROCOSS(4,2):RETURN
2590PROCOSS(4,1):PROCOSP(4,0,1):PROCRSS(1,9):RETURN
```

Only one object (4) can be lit (2550); if the player's not holding it, say so. All good negative thinking. At line 2560 we're still negative; is there some reason why the torch cannot be lit? Yes there is. First, it might be lit already, or have been lit before and now be out. So we say so, using Z% to choose the message. We might not be in the right room (2570). By 2580 all the conditions for success have been checked. We jot down the oiliness of the torch in Z%, tell the player it is lit (message 128) and what happens to it. In the case of a non-oily torch, we reset it to the stump state and exit. Otherwise (2590) we set it to its burning state; set it as a light source; and set the player on fire (state of room 1

equals 9). This will give the player eight turns to get to the misty area and drop his torch first, which is enough and to spare. The relevant messages are (notice how we use 130 to notify the player what happened to him as well):

125
It's already lit!

126
It's too damp to light.

127
There's nothing to light it with here.

128
(switches 129, 130) You light the wood at the brazier.

129
It flares up, but rapidly burns down to a black stump.

130
The oily wood catches fire and burns smoothly, making a fine torch. Alas, your clothes are also soaked in oil and you catch fire too!

11) Look

```
2630PROCRSP(R,1,0):RETURN
```

Just set room R to not VISITED, thus guaranteeing a description between-turns.

12) Q, Quit, Stop

```
2650F%=2:RETURN
```

The F% flag will make the program quit when we leave the GOSUB.

GOSUB.

13) Save

```
2680PROCINOUT(2)
2690RETURN
```

Having gone to such trouble with PROCINOUT, we can now use it. The argument '2' indicates that a save is occurring. The player will be prompted with a message which we'll write when putting the whole program together, next section.

14) Say

```
2710PRINT"OK, '";Y$;"'!":RETURN
```

Need I say more? It's a useless command.

15) Score

```
2730J%=0
2740FORI%=7TO14:IFFNP(I%,2)ANDFNR(I%)=1J%=J%+10
2750NEXT:J%=J%-10*(FNRP(19,1)+FNRP(22,1)+FNRP(16,1)+FNRP(27,1))
2760PRINT"You have scored ";STR$(J%);" out of 80":RETURN
```

This is quite subtle, as we'll see shortly. At 2730 we set the score counter, J%, to zero. We then scan the objects (2740) to see which of them is treasure (property 2) and held by the player; each time, we give the player 10 points. We need only scan objects 7 to 14 because none of the others are treasure; alternatively, we could drop TREASURE as a property and just tot up the three values concerned. If you had a game with 20 treasures, the method here is rather preferable! Then, at 2750, we give the player 10 points for each of 4 difficult rooms to reach: 19, 22, 16, and 27 (because FNRP's value will be -1 if that room is visited). We then tell the player his score. A message would have difficulty in substituting J%'s value in unless we rewrote the system, so we just do a print here.

If you're awake, you'll have noticed the maximum score is 70, not 80. We shall dole out the last 10 points for solving the endgame, and re-use line 2760 into the bargain.

16) Throw

```
 2780IFFNR(0%)<>1PROCM(110):RETURN
 2790IFFNRP(R,3)PROCM(131):PROCR(0%,37-R):RETURN
 2800IFR=27ANDFNS(13)AND0%=9PROCM(132):PROCOSS(13,0):PROCR(13,0):
PROCR(14,R):PROCR(9,0):RETURN
 2810IFR=31AND0%=4PROCM(134):J%=80:GOSUB2760:END
 2820GOTO2190
```

The nice thing about 'THROW' is its similarity to 'DROP', which means that we can use the DROP program again. The only special things about THROW which don't apply to DROP are THROW uttered from a position either side of the chasm, THROW PILUM (to handle the lion), and THROW TORCH (to win the endgame). So we must first check if it is carried (2780). If the room has property JUMPSPEC we tell the player, and move the object to room 37 – R. At 2800, we handle pilum-throwing. We check the room, the state of the lion (13), and the object thrown (9) – again, one of

these is unnecessary if you think about it. Assuming all is well, we tell the player what happens, reset the lion's state, move it and the pilum to destroy, move the wreath to the arena, and quit. (You should check what happens if one of these conditions isn't satisfied – for example, by throwing the pilum in the shop).

Line 2810 does the torch-throwing in room 31, the Senate. If all is well, we tell the player how clever he is, set J% to 80, and borrow the last line of the score program to print that he's scored 80 out of 80, and end the game.

Finally, if none of these special cases apply, THROW = DROP, and we go off to the DROP program (with another hanging GOTO). Some of the checks will be done again, but that doesn't matter. The relevant messages are:

131
You fling it across the chasm to the other side.

132
You hurl the pilum, killing the lion dead. It is dragged away, to waves of applause. A gold wreath is thrown to your feet by Caesar himself, who calls to you "Leave by the NEW exit!"

133
(this got out of order in programming, and is used in WEAR below)

134
You hurl the torch into the pool. It sputters and dies, plunging the room into darkness. You grab Ganopus' knife in the dark and use it on him before making your escape. You still have your treasures! You've won!

17) Wait

All we need is a simple 'OK'; we already have one at 2130, followed by a RETURN. So 2130 is the line we need; end of problem.

18) Wear

```
2840IF0%<>2THEN3000ELSEIFFNS(2)PROCM(133):RETURN
2850IFFNR(2)<>1PROCM(110):RETURN
2860PROCOSS(2,1):GOTO2130
```

If the object isn't the helmet – object 2 – wearing is impossible. If its state is not zero, it's worn already. If (2850) it isn't held, say so. Otherwise (2860) set the state to 1, print OK, and return.

133
You're already wearing it!

19) Zzm – magic move

```
2870IFFNR(0%)R=FNR(0%):RETURN ELSE3000
```

As long as O%'s room is not zero, move the player there, else tell him it's impossible.

20) Zzt – magic take

We scan the 'GET' command, and discover that 2060 is checking if the object is in the player's room. But 2070 is checking if the object is untakeable – even with magic takes, we don't want to take an untakeable object! So ZZT can join in at 2070. Should we have programmed GET in the wrong order, we can easily re-order the BASIC to suit the magic take requirements.

7.8 Assembling the program

Now we come to the part we've all been waiting for – assembling the program.

The first thing to do is to decide what type of machine you're writing for. Are we dealing with a cassette-based BBC Micro or Electron, or a disc-based BBC Micro? For each, PAGE, the bottom of the program, is set differently. On a cassette-based system, it is usually &E00 in hexadecimal. On a disc-based system, it's normally &1900, though we can cut that down. With Econet or Teletext, PAGE will be higher still. Where your program starts will define where your database starts.

For the purposes of this book, I've tried to be all things to all men! Rather than start at &E00, which will suit most readers but annoy those with discs who'll have to download the program to make it work, I've tried to choose a value of PAGE which will suit everybody. The advantage of this is that anybody can run the game – rather useful if you have commercial instincts! However, there is no complete solution to this problem. The ideal would be a program which

worked at any value of PAGE, like most BASIC programs; but whereas this would work with 'ROMAN', which is small enough, it certainly won't work with a bigger, commercial size game.

The relevant value of PAGE, which doesn't appear in the manuals, is &1100. This will allow *LOAD and *SAVE, for example. The disc operating system (on the BBC; at the time of writing no decision has been made about Electron disc filing systems) only uses &1100 to &1900 when writing individual bytes with BPUT.

OK, so for now we'll use &1100, but for the moment ignore the fact and simply type in the program as we have it.

On the way, you'll find various gaps and question marks, which now need to be filled in. Some we can do now, some in a little while. The order you fill these in doesn't really matter. For example, we can do much of line 10 now:

```
10NC=43:NT=27:NS=12:N%=????:Z=&C00
```

because we know how much vocabulary we have. We need to create an 'I don't understand that!' message for 330 and 350:

```
330D%=K%:IFD%=0ANDY$<>""PROCM(135):UNTIL FALSE
350PROCW(Y$,2):O%=J%:IFJ%>0ELSEPROCW(Y$,3):S%=J%:
IFJ%=0ANDD%<>3PROCM(135):UNTIL FALSE
```

135
I don't understand that!

and can use message 102 ('You can't do that!') in line 360:

```
360IFD%=1ANDS%>0PROCM(102):UNTIL FALSE
```

Also, we can complete lines 1000 (death) and 1005 (new game), using two suitable messages and the scoring program to tell the player how well he did:

```
1000DEFPROCDIE:PROCM(136):GOSUB2730:PROCNEWGAME
1005DEFPROCNEWGAME:PROCM(137):INPUTX$:IF(ASC(X$)OR32)<>110RUN ELSEEND
```

136
The gods welcome you to Hades (i.e. you're dead!)

137
Would you like another game?

The first line of PROCINOUT also needs a message.

We'll use one which will do for loading and saving, message 138:

```
5960DEFPROCINOUT(I%):PROCM(138):INPUTY$
```

138
Type the filename

and we also need a 'story so far' for line 30:

```
30PRINT'':PROCM(139)
```

139
You are a poor Roman and owe money to a swindler called Ganopus. He insists that you must pay him today, March 15th, by finding three valuable objects: one of gold, one of silver, and one of bronze. "Bring these to me in the Senate after dark and I'll forget your debt," he assures you. It is now noon. . .

We can also fill in the GOSUB calls in line 510:

```
  5100NC%GOSUB2000,2050,2170,2250,2300,2360,2380,2400,2450,2550,2630,2650,
2680,2710,2730,2780,2130,2840,2870,2070
```

You will have noticed that the program is rather short on spaces. This isn't vital in this game, but spaces do become important when you have a big game to fit into a similar space: they both slow the response and take up room. The decision is yours. If you like legibility is of prime importance, use spaces, but if you want a big game, don't use spaces. The player probably won't mind either way!

So having typed it, we save the program onto tape or disc depending on your system. Then, assuming that you want to follow my method, type

```
PAGE = &1100
```

(If you use a cassette system and don't want to bother, you will have to subtract – or get the computer to subtract – &300 from all my numbers henceforth!) Now type

```
LOAD "ROMAN"
```

followed by

```
PRINT ~TOP
```

to get TOP in hexadecimal – because now you're going to have to do a little in hexadecimal, whether you like it or not. When I first programmed this, I got 256B for the value. This is extremely likely to increase, because

more program lines will be required to solve errors (in fact, I used about 150 extra bytes).

Now we shall need about 1000 bytes for BASIC workspace; you can get away with about half of this, but it doesn't hurt to be on the safe side. So ask BASIC to add 1000 to TOP, and tell you the answer in hexadecimal. You might as well round it up to an easy number. I got &2A00.

So if HIMEM is set just under this, the objects part of the database can begin at &2A00. So, load in 'DATAGEN' (your regular PAGE will do fine). Run it, and answer 'O' for Objects, and '&2A00' for the value of o%. Continue to feed in all the 15 objects, ending with a return to terminate the program.

You now have a choice. If you are supremely confident of your ability to type in perfect data all evening, continue immediately with the rooms. If, like me, you are more cautious, you should now save the objects onto tape/disc. The information you need is that last value of P%, which points to where the rooms will be going. This should be &2A50 (16 objects, counting object zero, times 5 bytes should also tell you this, with some arithmetical help from the computer). So you act cowardly, and save all the object data to a file named 'ODATA' by typing:

```
*SAVE ODATA 2A00 2A50
```

Now continue the procedure. Rerun DATAGEN, this time for the rooms. Answer '&2A50' for r%, since that's immediately after the objects. Feed in all the room data, again finishing with return. That should leave P% as &2D50, which is where the verbs will start. Save the rooms away with

```
*SAVE RDATA 2A50 2D50
```

(If nothing disastrous has happened, the object data will still be in memory. You could save the two together if you preferred.)

Next, do the vocabulary, starting with verbs. Run the program for the verbs, answering '&2D50' for v%. At the end of the verbs, P% should be &2E52; again, only a matter of arithmetic really! Save these:

```
*SAVE VDATA 2D50 2E52
```

and continue with the things (t% = &2E52), ending

with P% = &2ED9. Put these away:

```
*SAVE TDATA 2E52 2ED9
```

and put in the specials (s% = &2ED9), ending with P% = &2F15. Save these also:

```
*SAVE SDATA 2ED9 2F15
```

Next, and mercifully last, come the messages. Start them with message 1, at &2F15. Type steadily. You can always halt the program and save what you have at any time, then continue from where you left off, with whatever value P% was last quoted. If you should then make a mistake, *LOAD the MDATA back to &2F15, and start up where you last saved from. That way you avoid having to re-type everything.

When you've got all that data in, save it with

```
*SAVE MDATA 2F15 4B6A
```

The last number was where my typing finished. Your typing may differ slightly, depending on where you put carriage returns in, and so on. Use your number, not mine!

We are now ready to assemble the whole program. Clear the computer, set PAGE = &1100, and 'LOAD "ROMAN"'. While you have it in front of you, you can attend to a few final details:

```
  9MODE7:HIMEM=&29FF
 10NC=43:NT=27:NS=12:N%=&2A03:Z=&C00
 11o%=&2A00:r%=&2A50:v%=&2D50:t%=&2E52:s%=&2ED9:m%=&2F15
5970$Z=Y$+" 2A00":IFI%=1$Z="L. "+$Z ELSE$Z="S. "+$Z+" 2D50"
```

In line 9, set HIMEM one under o% (get the computer to do the sum if you are uneasy with hexadecimal). In line 10, we need a value for N% to store the number of turns the player has had. We here use the short message number for object zero, which of course isn't being used – we could have used any of the other bytes between &2A00 and &2A04 inclusive. In line 11, fill in the values you used for o%, r%, v%, t%, s%, and m%. Finally, in line 5970 – part of PROCINOUT – fill in the start address for the objects (o%, equal to &2A00) and the finish address for the rooms (&2D50, equal to v%). Thus PROCINOUT will load/save just the objects and rooms, as desired.

Now *LOAD in the objects, rooms, verbs, things, specials, and messages to their appropriate places in memory:


```
*LOAD ODATA 2A00
*LOAD RDATA 2A50
*LOAD VDATA 2D50
*LOAD TDATA 2E52
*LOAD SDATA 2ED9
*LOAD MDATA 2F15
```

Two last actions, and we're set. We have to make the room of object zero be the player's room, because that's where the playing program expects to find it. Just type

```
?&2A02 = 2
```

which sets the value of byte &2A02 (room of object zero) to be 2. The other action is to set ?N% to be zero, for the turn counter. Type

```
?&2A03 = 0
```

Now again, play it safe. Type

```
*SAVE ROMAN 1100 4B6A
```

which will save both the playing program and the database, calling the result 'ROMAN' once more. You may not have realised, but when you perform a regular SAVE, what happens is *SAVE name 'PAGE' 'TOP', where BASIC substitutes the values. So there is nothing wrong with *SAVEing a program.

We also need to create an initial database file to start the player off. Of course, the position in memory is the initial position. Now type

```
*SAVE INIT 2A00 2D50
```

which will store objects and rooms – which contain the entire dynamic part of the database – in a file called INIT. You might think that PROCINOUT(2) would suffice. But it won't because until m% is initialised, we can't print the messages. If you prefer, though, set m% = &2F15 in immediate mode and try PROCINOUT (2), which should work.

7.9 On debugging

This program should work first time. But it didn't when I first programmed it. I kept a list of the bugs I found as I went along – there were 25 at the final count. One can never be sure such programs are totally bug-free. . .

It's worthwhile looking at some of my errors, because often the fear of making a lot of errors on a big program is what stops people from writing a program! Seldom are errors fatal, especially if you remember to *SAVE once in a while as you go along.

My first bug came immediately after RUN. I'd forgotten to set a variable while I was creating the database system. Since I knew which line number it was, thanks to the error message, it was only a question of printing out the value of each variable mentioned in the line until I found an undefined one.

The program now ran, only I got no room description. I escaped, and printed R, the room number. It was zero, not 2. A quick check convinced me I'd forgotten to load object zero's room to be 2 ('PRINT FNR(0)'). I was tempted to set it by 'PROCR(0,2)' until I remembered that INIT is pulled in every game, and it's INIT that needed setting. In fact, it's good practice to set the main program's initial database as well, even though that never gets used. If you should make a blunder and lose INIT, you've got a back-up copy in the main program. So we PROCINOUT(1), giving INIT as the filename, and then PROCR(0,2), followed by a cautious PRINT FNR(0) to be certain. Then PROCINOUT(2) to put INIT away again.

This time I got the wrong initial message because I had mistyped m%'s value in the program? Then INV wouldn't work, because of a stray 'I' when I meant I%...Note that I have yet to leave the first room. Simply by testing INV, LOOK, etc., you have tested a lot of code.

Then came line 610. I had missed out a vital space, and so created a nonexistent variable. None of these bugs had necessitated a resave yet. I then tried 'OOH' to see how my parser handled a word it didn't understand. The program collapsed, owing to an infinite loop I'd misprogrammed in PROCW. 'TRACE ON' soon found that one.

Then I ventured into the shop. No shopkeeper! I escaped and checked the state of the shop. It was 1, as it should be. So the message was wrong. This called for a little investigation of the message structure itself. First, I needed the message before 45, the faulty message. So I typed 'PROCM(44)' and received, as expected, 'You're in the street.' The point of doing this was to set P% to point at the first byte of message 45, the one I was interested in. Now in immediate mode I typed:

```
FOR I% = 0 TO 60: PRINT I%, P%?I%: NEXT
```


with CTRL/N on, which printed out the contents of P% and the next 60 bytes, all numbered. If you think about the structure, what we should see is a lot of ASCII codes for the 'You are in an old shop . . .', then a 13 (carriage return for the end of the first line) then some more ASCII – the message takes up two lines – and another 13. Next came the switches, beginning at I% = 51. These ought to have been – remember two bytes per switch, low byte first – 0, 0, 46, 0. They weren't. They read 46, 0, 0, 0. I'd put the switches in the wrong order! Now it was necessary to be methodical. I reloaded INIT, putting the game into a pristine state. Then I typed

```
P%?51=0
P%?53=46
```

which changed around the two message switches. I then did a *SAVE of the whole program again.

Several more times I had to dive into the database and hunt up something silly. But most of the time it was just wrong BASIC! I found eventually, on using magic moves to test out the priest that I had switched the lion and priest in postprog: apparently I would die from the lion in the temple, and from the priest in the arena! You must test exhaustively, and not just by giving the program the instructions you expect it to accept.

Sometimes the logic was wrong. I forgot to check if chickens were held before they were killed . . . I forgot to leave any exit from the first maze into the arena . . . I somehow had defined the mouse as a light source . . . one room description said 'northeast' when I meant 'northwest' . . . picking up a dead chicken still provoked 'the chicken flutters away', which kept me in giggles for some little time . . . and so on.

One problem you'll have to watch out for as your games get larger is the 'No Room' message. You can monitor how your space is going by sticking in the

```
PRINT ~ !2 AND &FFFF
```

around line 100. This will print out the top of BASIC's work space at the moment. It'll fill up gradually as the program uses all its procedures, and will then steady down. You print in hexadecimal, by the way, because you know o% in hexadecimal; for no other reason.

7.10 A listing of the (non-database part of) 'ROMAN'

It's probably worthwhile to see the entire BASIC part

of the program in one listing, so here it is:

```
  8REM ONERRORGOTO1005
  9MODE7:HIMEM=&29FF
 10NC=43:NT=27:NS=12:N%=&2A03:Z=&C00
 11O%=&2A00:R%=&2A50:V%=&2D50:T%=&2E52:S%=&2ED9:M%=&2F15
 15Z$=STRING$(20," ")
 20FORI%=7TO8:PRINTTAB(10,I%);CHR$(141);"Roman Adventure":NEXT
 30PRINT'':PROCM(139)
 40PRINT'':PROCINOUT(1):PRINT''
 50Q%=0
100REPEAT R=FNR(0):IFR<>Q% ORNOTFNRP(R,1)PROCDR(R)
110PROCRSP(R,1,1):Q%=R:F%=0
200REPEAT
210IFF%=0REPEAT INPUT'":"Z$:UNTILZ$<>""
220PROClc
230J%=INSTR(Z$," "):IFJ%=0X$=LEFT$(Z$,4):Y$="":GOTO300
240X$=LEFT$(LEFT$(Z$,J%-1),4):Y$=RIGHT$(Z$,LEN(Z$)-J%)
250IFLEFT$(Y$,1)=" "REPEATY$=RIGHT$(Y$,LEN(Y$)-1):UNTIL LEFT$(Y$,1)<>'
260Y$=LEFT$(Y$,4)
300PROCW(X$,1):C%=J%:IFJ%>0THEN330
310Y$=X$:PROCW(Y$,2):D%=J%:PROCW(Y$,3):IFD%+J%=0PRINT"EH?":UNTIL FALSE
320PRINT"What do you want to do with the ";Y$;"?":INPUTZ$:PROClc:
X$=LEFT$(Z$,4):PROCW(X$,1):C%=J%:IFJ%=0PRINT"EH?":UNTIL FALSE
330D%=K%:IFD%=0ANDY$<>""PROCM(135):UNTIL FALSE
3400O%=0:S%=0:IFY$=""THEN360
350PROCW(Y$,2):O%=J%:IFJ%>0ELSEPROCW(Y$,3):S%=J%:
IFJ%=0ANDD%<>3PROCM(135):UNTIL FALSE
360IFD%=1ANDS%>0PROCM(102):UNTIL FALSE
370IFD%>0ANDD%<3ANDY$=""PRINTX$;" what?":INPUTZ$:PROClc:
Y$=LEFT$(Z$,4):GOTO330
380UNTIL TRUE
400Y%=FNL
410FORI%=11TO13STEP2:IFR=FNR(I%)PROCOSS(I%,1)
420NEXT
500F%=0
510ONC%GOSUB2000,2050,2170,2250,2300,2360,2380,2400,2450,2550,2630,2650,
2680,2710,2730,2780,2130,2840,2870,2070
520IFF%=9PROCDIE
530IFF%=1Z$=Y$:GOTO200
540IFF%=2PROCNEWGAME
600PROCR(0,R)
610IFY%=0ANDNOTFNL ANDR<>Q%PROCM(93):PROCDIE
620I%=?N%+1+(?N%=255):?N%=I%:IFI%=35PROCM(94)
630IFI%=50PROCM(95):FORI%=2TO22:PROCRSP(I%,0,0):NEXT:PROCRSP(31,0,0)
640I%=FNRS(1):IFI%=0ELSEPROCRSS(1,I%-1):PROCM(96):
IFI%=1PROCM(97):PROCDIE
650IFFNS(13)PROCM(98):PROCDIE
660IFFNS(11)PROCM(99):PROCDIE
670IFR<>31ELSEI%=FNRS(R)-1:PROCRSS(R,I%):IFI%=0PROCM(100):PROCDIE
680UNTIL FALSE
1000DEFPROCDIE:PROCM(136):GOSUB2730:PROCNEWGAME
1005DEFPROCNEWGAME:PROCM(137):INPUTX$:IF(ASC(X$)OR32)<>110RUN ELSEEND
2000L%=FNRL(R)+D%-2:IF?L%=0PROCM(101):RETURN ELSEJ%=L%?1:G%=0
2010IFJ%>00NJ%GOSUB6000,6020,6060,6090,6120,6170,6230,6260,6280,6300
2020IFF%=90RG%=1RETURN ELSER=?L%:RETURN
2050IFS%>0THEN3000
2060IFR<>FNR(0%)PROCM(103):RETURN
2070IFFNP(0%,1)PROCM(104):RETURN
2080IFFNR(0%)=1PROCM(105):RETURN
2090IFO%=2AND FNRS(R)ANDR=4PROCM(106):RETURN
2100IFO%<>3ELSEIFFNR(15)=1PROCM(107):PROCR(15,0):PROCR(3,1):RETURN
ELSEPROCM(108):RETURN
2110IFO%=8ANDFNS(8)=0PROCM(109):RETURN
2120PROCR(0%,1)
```



```
2130PRINT"OK":RETURN
2170IFS%>0THEN3000
2180IFFNR(0%)<>1PROCM(110):RETURN
2190IFFNRP(R,2)PROCM(111):PROCR(0%,0):RETURN
2200IFO%=2PROCOSS(2,0)
2210IFO%=3ANDR=FNR(8)PROCM(112):PROCR(8,1):PROCR(3,R):RETURN
2220PROCR(0%,R):GOTO2130
2250IFNOTFNP(0%,4)THEN3000
2260IFFNR(0%)<>1PROCM(110):RETURN
2270PROCR(0%,0):PROCM(113):RETURN
2300IFS%=20%=10
2310IFFNR(0%)<>1PROCM(110):RETURN
2320IFO%<>10THEN3000
2330IFR<>50RFNRP(R,0)PROCM(114):RETURN
2340R=23:PROCM(115):PROCR(10,0):RETURN
2360F%=1:IFY$=""PRINTX$;" where?":INPUTY$
2370RETURN
2380PROCDR(1):RETURN
2400IFNOTFNRP(R,3)PROCM(116):RETURN
2410IFFNR(5)=10RFNR(9)=1F%=9:Z%=0ELSEZ%=1:R=37-R
2420PROCM(117):RETURN
2450IFS%>00RNOTFNP(0%,3)THEN3000
2460IFR<>FNR(0%)ANDFNR(0%)<>1PROCM(103):RETURN
2470IFO%=11THEN2510ELSEIFO%=13THEN2520ELSEIFO%=8ELSEPROCM(120):RETURN
2480IFFNR(8)<>1PROCM(110):RETURN ELSEIFFNR(5)<>1PROCM(121):RETURN
2490PROCOSS(8,1):PROCOSP(8,3,0):PROCM(122)
2500IFR<>FNR(11)RETURN ELSEPROCOSS(11,0):PROCR(11,0):PROCR(5,0):
PROCM(123):RETURN
2510PROCM(124):F%=9:RETURN
2520IFFNR(9)<>1PROCM(121):RETURN ELSE0%=9:GOTO2800
2550IFO%<>4THEN3000ELSEIFFNR(4)<>1PROCM(110):RETURN
2560Z%=FNS(4):IFZ%PROCM(124+Z%):RETURN
2570IFR<>4PROCM(127):RETURN
2580Z%=-FNP(4,5):PROCM(128):IFZ%=0PROCOSS(4,2):RETURN
2590PROCOSS(4,1):PROCOSP(4,0,1):PROCRSS(1,9):RETURN
2630PROCRSP(R,1,0):RETURN
2650F%=2:RETURN
2680PROCINOUT(2)
2690RETURN
2710PRINT"OK, '";Y$;"'!":RETURN
2730J%=0
2740FORI%=7TO14:IFFNP(I%,2)ANDFNR(I%)=1J%=J%+10
2750NEXT:J%=J%-10*(FNRP(19,1)+FNRP(22,1)+FNRP(16,1)+FNRP(27,1))
2760PRINT"You have scored ";STR$(J%);" out of 80":RETURN
2780IFFNR(0%)<>1PROCM(110):RETURN
2790IFFNRP(R,3)PROCM(131):PROCR(0%,37-R):RETURN
2800IFR=27ANDFNS(13)AND0%=9PROCM(132):PROCOSS(13,0):PROCR(13,0):
PROCR(14,R):PROCR(9,0):RETURN
2810IFR=31AND0%=4PROCM(134):J%=80:GOSUB2760:END
2820GOTO2190
2840IFO%<>2THEN3000ELSEIFFNS(2)PROCM(133):RETURN
2850IFFNR(2)<>1PROCM(110):RETURN
2860PROCOSS(2,1):GOTO2130
2870IFFNR(0%)R=FNR(0%):RETURN ELSE3000
3000PROCM(102):RETURN
5080DEFFNOL(0%)=0%*5+o%
5090DEFFNRL(R%)=R%*24+r%
5100DEFFNR(0%)=?(FNOL(0%)+2)
5110DEFFNS(0%)=?FNOL(0%)
5130DEFFNP(0%,P%):LOCALI%:I%=?(FNOL(0%)+1)AND2^P%:=(I%>0)
5140DEFPROCOSP(0%,P%,I%)
5150A%=FNOL(0%):IFI%=0A%?1=A%?1AND(&FF-2^P%)ELSEA%?1=A%?1OR2^P%
5160ENDPROC
5170DEFPROCOSS(0%,I%):?FNOL(0%)=I%:ENDPROC
5200DEFPROCR(0%,R%):?(FNOL(0%)+2)=R%:ENDPROC
5220DEFPROCDO(0%)
```

```
5230M%=?(FNOL(O%)+4+(FNR(O%)=1))
5240Z%=FNS(O%):PROCM(M%):ENDPROC
5310DEFFNRS(R%)=?FNRL(R%)
5320DEFFNRP(R%,P%):LOCALI%
5330I%=?(FNRL(R%)+1)AND2^P%:=(I%>0)
5340DEFPROCRSP(R%,P%,I%)
5350A%=FNRL(R%)+1:IFI%=0?A%=?A%AND(&FF-2^P%)ELSE?A%=?A%OR2^P%
5360ENDPROC
5370DEFPROCRSS(R%,I%):?FNRL(R%)=I%:ENDPROC
5440DEFFNL:LOCALI%,J%
5450IFFNRP(R,0)THEN=TRUE
5460I%=FALSE:FORJ%=1TO15:IFFNP(J%,0)AND(FNR(J%)=1ORFNR(J%)=R)I%=TRUE:
J%=15
5470NEXT:=I%
5520DEFPROCDR(R%):LOCAL I%,J%
5530IFFNL OR R%=1 ELSEPRINT"It is pitch dark":ENDPROC
5540M%=?(FNRL(R%)+23+FNRP(R%,1))
5550Z%=FNRS(R%):PROCM(M%)
5560J%=TRUE
5570FORI%=1TO15:IFFNR(I%)=R% PROCDO(I%):J%=FALSE
5580NEXT
5590IFJ% AND R%=1 PRINT"Nothing"
5600ENDPROC
5610DEFPROCM(M%):LOCALI%
5620IFM%=0ENDPROC ELSE P%=m%:I%=1
5630A%=?P%AND&F:B%=?P%DIV16:P%=P%+1
5640IFI%=M%THEN5670
5650FORD%=1TOA%:P%=P%+LEN$P%+1:NEXT
5660P%=P%+2*B%:I%=I%+1:GOTO5630
5670IFA%=0P%=P%+1:GOTO5710ELSEFORD%=1TOA%
5680PRINT $P%:P%=P%+LEN$P%+1
5690NEXT
5710IFB%=0ENDPROC
5720M%=Z%:IFM%>(B%-1)M%=B%-1
5730M%=P%!(2*M%)AND&FFFF:PROCM(M%)
5740ENDPROC
5750DEFPROCW(X$,I%):J%=0:K%=0:U%=1:ON I% GOTO 5760,5765,5770
5760H%=NC:P%=v%:J%=6:GOTO5780
5765H%=NT:P%=t%:J%=5:GOTO5780
5770H%=NS:P%=s%:J%=5
5780IFX$<FNSTR(U%,I%)ORX$>FNSTR(H%,I%)J%=0:ENDPROC
5790IFX$=FNSTR(U%,I%)M%=U%:PROCSET:ENDPROC
5800IFX$=FNSTR(H%,I%)M%=H%:PROCSET:ENDPROC
5810REPEAT IF(H%-U%)=1J%=0:K%=0:UNTIL TRUE:ENDPROC
5820M%=(U%+H%)DIV2:IFX$=FNSTR(M%,I%)UNTIL TRUE:PROCSET:ENDPROC
5830IFX$>FNSTR(M%,I%)U%=M%:UNTIL FALSE ELSE H%=M%:UNTIL FALSE
5850DEFPROCSET:P%=P%+(M%-1)*J%+4:J%=?P%:IFI%=1THENK%=P%?1ELSEK%=1
5860ENDPROC
5880DEFFNSTR(M%,I%)
5890IFI%=1T%=v%+6*(M%-1)ELSEIFI%=2T%=t%+5*(M%-1)ELSET%=s%+5*(M%-1)
5900!Z=!T%:Z?4=13
5910IFRIGHT$($Z,1)="  "REPEAT$Z=LEFT$($Z,LEN($Z)-1):UNTIL RIGHT$($Z,1)
<>" "
5920=$Z
5940DEFPROClc:FORA%=1TOLENZ$:B%=ASC(MID$(Z$,A%,1))OR32
5950Z$=LEFT$(Z$,A%-1)+CHR$B%+RIGHT$(Z$,LENZ$-A%):NEXT:ENDPROC
5960DEFPROCINOUT(I%):PROCM(138):INPUTY$
5970$Z=Y$+" 2A00":IFI%=1$Z="L. "+$Z ELSE$Z="S. "+$Z+" 2D50"
5980X%=0:Y%=&C:CALL &FFF7:ENDPROC
6000IFFNR(7)<>1ORFNR(12)<>1ORFNR(14)<>1PROCM(76):F%=9:RETURN ELSE
PROCM(77):RETURN
6020I%=FNRS(3):IFI%=0RETURN
6030PROCM(77+I%):IFFNR(4)=1PROCOSP(4,5,1)
6040PROCRSS(3,I%-1):RETURN
6060PROCRSS(R,0):RETURN
6090PROCM(82+FNRP(R,0)):RETURN
```



```
6120IFFNR(2)<>1PROCM(83):G%=1:RETURN
6130IFFNS(2)=0PROCM(84):G%=1:RETURN
6140PROCM(86):RETURN
6170IFFNRS(R)RETURN
6180IFFNRS(1)=0F%=9:PROCM(87):RETURN
6190PROCM(88):PROCRSS(1,0):PROCRSS(R,1)
6200IFFNR(4)=1ANDFNS(4)=1PROCOSS(4,2):PROCOSP(4,0,0):PROCM(89)
6210RETURN
6230IFR<>FNR(6)ORFNS(15)RETURN
6240PROCR(15,R):PROCOSS(15,1):RETURN
6260IFFNR(2)=1F%=9:PROCM(90):RETURN ELSERETURN
6280PROCRSP(R,1,0):RETURN
6300PROCM(92):RETURN
```

7.11 The input for 'DATAGEN'

It's also useful to see the 'model' input to 'DATAGEN' which produced the version of the database I used. I've chopped it up into the appropriate sections. Where a blank line appears, a carriage return only was typed.

1) Objects – at &2A00

```
1, 1
0
0
1
10
4
0
1
2, 2
0
10
4
2
4
3,3
0
3
4
10
6
5
6
4,4
0
10
8
7
11
5,5
0
10
19
15
16
6,6
0
10
19
17
18
7,7

0
2
10
17
19
20
8,8
0
3
4
10
15
21
24
9,9
0
10
15
27
28
10,10
0
10
20
29
30
11,11
0
1
3
10
21
0
31
12,12
0
2
10
22
32
33

13,13
0
1
3
10
27
0
34
14,14
0
2
10
0
35
36
15,15
0
4
10
0
37
38
0
```

2) Rooms – at &2A50

```
1, ROOM 1
0
0
10

0
39
2, ROOM 2
0
0
10
W
3,0

40
41
3, ROOM 3
2
0
10
E
2,0
N
31,1
S
4,0
W
5,2

42
43
4, ROOM 4
1
0
10
N
3,3

44
45
5, ROOM 5
0

0
10
W
5,4
NW
6,0
SW
12,0
E
3,2

47
47
6, ROOM 6
0
0
10
W
7,0
SE
5,0

48
48
7, ROOM 7
0
0
10
W
8,0
E
6,0
NE
20,5

49
50
8, ROOM 8
0
0
10
E
```

```
7,0
SW
9,0

51
51
9, ROOM 9
0
0
10
NE
8,0
SE
10,0
W
18,0

52
52
10, ROOM 10
0
0
10
NW
9,0
E
11,0
SW
16,6

53
53
11, ROOM 11
0
0
10
W
10,0
E
12,0
SE
13,0

56
56
12, ROOM 12
0
0
10
NE
5,0
W
11,0

57
57
13, ROOM 13
0
0
10
NW
11,0
S
14,0

58
58
14, ROOM 14
0
0
10
N
13,7
S
15,7

59
60
15, ROOM 15
0
0
10
N
14,0

61
61
16, ROOM 16
0
0
10
NE
10,0
S
17,0

62
63
17, ROOM 17
0
0
10
N
16,0

64
64
18, ROOM 18
0
0
3
10
E
9,0

65
66
19, ROOM 19
0
0
3
10

67
67
20, ROOM 20
0
0
10
NW
21,8
U
21,8
SW
7,5

68
69
21, ROOM 21
0
0
10
W
22,0
SE
20,0
D
20,0

70
71
22, ROOM 22
0
0
10
E
21,0

72
72
23, ROOM 23
0
0
2
10
E
23,9
NE
24,0
NW
24,0
SW
24,0

73
73
24, ROOM 24
0
0
2
10
NW
23,0
N
25,0
SE
23,0
NE
24,9

73
73
25, ROOM 25
0
0
2
10
SW
26,0
S
23,0
SE
25,9
NW
24,0

73
73
25,9
NW
24,0

73
73
26, ROOM 26
0
0
2
10
W
27,0
SW
23,0
SE
24,0
NE
25,0

73
73
27, ROOM 27
0
0
10
SW
28,0

74
74
28, ROOM 28
0
0
2
10
N
29,0
NE
28,9
E
28,9
SE
28,9
S
28,9
SW
28,9
W
28,9
NW
28,9
73
73
29, ROOM 29
0
0
2
10
E
30,0
N
28,0
NE
28,0
SE
28,0
S
28,0
SW
28,0
W
28,0
NW
28,0

73
73
30, ROOM 30
0
0
2
10
W
9,10
N
28,0
NE
28,0
E
28,0
SE
28,0
S
28,0
SW
28,0
NW
28,0
73
73
31, ROOM 31
2
0
10

75
75
0
```


3) Verbs – at &2D50

```
catch   pay
2,2     5,2
d       q
1,22    12,0
down    quit
1,22    12,0
drop    run
3,2     6,3
e       s
1,8     1,12
east    sacrifice
1,8     9,2
eat     save
4,1     13,0
get     say
2,2     14,3
give    score
5,2     15,0
go      se
6,3     1,10
inv     south
7,0     1,12
jump    stop
8,3     12,0
kill    sw
9,2     1,14
light   take
10,1    2,2
look    throw
11,0    16,1
move    u
6,3     1,20
n       up
1,4     1,20
ne      w
1,6     1,16
nort    wait
1,4     17,0
nw      wear
1,18    18,1
        west
        1,16
        zzm
        19,1
        zzt
        20,1
```

4) Things – at &2E52

```
braz
1
bronze
12
bust
12
cat
3
chicken
8
coin
10
contrap
6
gladius
5
gold
14
helmet
2
javelin
9
lion
13
money
10
mouse
15
pilum
9
priest
11
ring
7
sesterc
10
silver
7
spear
9
stump
4
sword
5
tabby
3
torch
4
trap
6
wood
4
wreath
14
coliseum
1
door
1
ganopus
1
guard
1
hole
1
man
2
oil
1
pool
1
shopkeep
1
slave
1
toll
2
```

5) Specials – at &2ED9

```
booth
1
```

6) Messages – at &2F15

These are laid out exactly as typed in, and fit fairly neatly onto 40-column modes.

```
0,1
There is a brazier of glowing coals
here.

2,2
A helmet

0
3
0,3
(which you are wearing)

0,4
A military helmet lies nearby.

0,5
A contented cat

0,6
A tabby cat frisks here.

3,7

8
9
10
0,8
An unlit torch

0,9
A burning torch

0,10
A blackened stump

3,11

12
13
14
0,12
A tapered piece of wood as long as
your arm lies here.

0,13
There is a burning torch here.

0,14
There is a black stump here.

0,15
A gladius

0,16
A vicious-looking gladius lies here.

0,17
A trap

0,18
There is a contraption of sharp iron,
wood and cheese here.

0,19
A ring

0,20
There is a silver ring, stolen by the
slave, here!

2,21

22
23
0,22
A chicken

0,23
A dead chicken

2,24

25
26
0,25
A chicken struts around, clucking.

0,26
A dead chicken lies sadly here.

0,27
A pilum

0,28
Someone has left a pilum here.

0,29
A sesterce

0,30
An old sesterce is on the floor.

0,31
A priest, his hands red with blood,
looks at you expectantly.

0,32
A bronze bust


0,33
A bronze bust of Cicero is yours for
the taking!

0,34
A roaring lion bears down on you, its
jaws agape!
```



```
0,35
A gold wreath

0,36
The gold wreath of victory is here!

0,37
A dead mouse

0,38
There is a dead mouse here.

0,39
You are carrying:

0,40
You're at home.

0,41
You're in your house, which is poor but
comfortable. To the west lies a
street.

0,42
You're in the street.

0,43
You are in a long east-west street.
To the north lies the Senate, and to
the south is a small shop.

0,44
You're in the shop.

2,45
You are in an old shop, with its exit
northwards.

0
46
0,46
A shopkeeper is keeping his eye on you.

0,47
You're east of the Coliseum, a large
circular building. A road around it
leads northwest and southwest. There
is a toll-booth west and a street
east.

0,48
You're northeast of the Coliseum.
The road goes west and southeast.

0,49
You're north of the Coliseum.

0,50
You're north of the Coliseum. The road
goes east and west, and an archway
leads northeast through some barracks.

0,51
You're northwest of the Coliseum; the
road goes east and southwest.

0,52
You're west of the Coliseum; the road
goes northeast and southeast. A
valley stretches west, and a closed
door bars the way east.

2,53
You're southwest of the Coliseum; the
road goes northwest and east.

54
55
0,54
To the southwest is a dank, misty area.

0,55
To the southwest is some waste ground.

0,56
You're south of the Coliseum; the
road leads east and west. A narrow
lane goes southeast.

0,57
You're southeast of the Coliseum; the
road leads northeast and west.

0,58
You're in a lane winding from northwest
to south.

0,59
You're in the farm.

0,60
This is a crude farm with little
furniture. Small holes dot the base
of many of the walls. Doors lead
north and south.

0,61
You're in an enclosed farmyard. The
only exit is north.

0,62
You're on the waste ground.

0,63
You are on some damp waste ground. A
cottage is south, and the road is to
the northeast.

0,64
You find yourself in a cottage, the
hideout of the slave. The only exit
is north.

0,65
You're in the valley.

0,66
You are in a valley curving from the
east and ending at a chasm to the
south. A gully is visible across the
chasm.
```

```
0,67
You're in a gully across the chasm,
with no obvious exits.

0,68
You're in the anteroom.

0,69
You find yourself in an anteroom to
the temple, to which steps lead up
to the northwest. A passage leads
back southwest through the barracks.

0,70
You're at the east end of the temple.

0,71
You are at the eastern end of an east-
west temple to Zeus. Stairs exit
down to the southeast.

0,72
You're at the west end of the temple.

0,73
You're in a maze of milling crowds,
here for the Games, and jostling you
about.

0,74
You are in the Coliseum arena,
surrounded by excited crowds. A
postern gate leads southwest out of
the arena.

0,75
You are in the Senate, a luxurious
area, but unlit at this late hour.
Ganopus meets you by the bathing pool.
Instead of taking your three
treasures, he treacherously draws a
knife and moves towards you to silence
you forever!

0,76
As you enter, a group of senators
leap on you, mistaking you for
Caesar. They plunge their daggers into
you as one man, before noticing
their sad error.

0,77
The conspirators have given up waiting
for Caesar and gone home.

1,78
You stride through a half-full patch
of oil.

80
1,79
You stride through a patch of oil.

80
0,80
You and your belongings are soaked.

0,81
The booth is closed, as the games
don't start till dark.

0,82
The man on the booth demands payment
and won't let you in otherwise.

1,83
The guards see you are not a soldier.

85
1,84
The guards see your helmet, but
realise you are not a soldier.

85
0,85
They bar your way.

0,86
The guards assume you are a soldier,
and let you pass.

0,87
In the mist, an escaped slave grabs
you and chokes you to death.

0,88
With a hiss, the mist condenses and
extinguishes you. You catch a glimpse
of a slave running away.

0,89
The mist also puts out your torch,
worse luck.

1,90
A priest appears. "No soldiers in
the temple!" he shouts.

91
0,91
You are rapidly arrested and executed.

0,92
To your relief, you fall out of a
door which is slammed behind you.

0,93
Thieves kill you while you flounder
around in the dark, unfortunately.

0,94
The sun is setting.

0,95
The sun has set.

0,96
You're on fire!

0,97
You burned to a crisp!

0,98
```



```
The lion grabs you and eats you.

1,99
The priest shouts "Imposter!"

91
0,100
Ganopus' knife slits your throat.

0,101
You can't go in that direction!

0,102
You can't do that!

0,103
That's not here!

0,104
You can't take that!

0,105
You're already holding that!

0,106
The shopkeeper won't let you!

0,107
With a loud miaow, the cat jumps
into your arms and scoffs the mouse.

0,108
The cat refuses to be picked up.

0,109
The chicken flutters away from you.

0,110
You're not holding that!

0,111
The object you dropped is soon lost
among the feet of the crowd.

0,112
The chicken, seeing the cat coming,
gives a squawk and flutters up. You
grab it and hold onto it.

0,113
It tastes awful, but you force it
down your throat.

0,114
Nobody seems interested in that!

0,115
You pay your sesterce and enter the
crowded Coliseum.

0,116
Whoopeee!

2,117
You leap across the chasm, and
scrabble for a handhold.

118
119
0,118
You're carrying something awkward,
which causes you to miss your hold
and fall to your death.

0,119
You find a hold and haul yourself
to safe ground.

0,120
You can't bring yourself to kill the
cat, alas.

0,121
You have no suitable weapon.

0,122
You slice its head off with the
gladius.

0,123
The priest nods approvingly, removes
your gladius with a prayer, and
leaves.

1,124
You attack the priest, who calls
loudly for help.

91
0,125
It's already lit!

0,126
It's too damp to light.

0,127
There's nothing to light it with here.

2,128
You light the wood at the brazier.

129
130
0,129
It flares up, but rapidly burns down
to a black stump.

0,130
The oily wood catches fire and burns
smoothly, making a fine torch. Alas,
your clothes are also soaked in oil
and you catch fire too!

0,131
You fling it across the chasm to the
other side.

0,132
You hurl the pilum, killing the lion
dead. It is dragged away, to waves of
applause. A gold wreath is thrown
to your feet by Caesar himself, who
calls to you "Leave by the NEW exit!"
```

```
0,133
You're already wearing it!

0,134
You hurl the torch into the pool. It
sputters and dies, plunging the room
into darkness. You grab Ganopus'
knife in the dark and use it on him
before making your escape.
You still have your treasures!
You've won!

0,135
I don't understand that!

0,136
The gods welcome you to Hades (i.e.
you're dead!)

0,137
Would you like another game?

0,138
Type the filename

0,139
You are a poor Roman and owe money
to a swindler called Ganopus. He
insists that you must pay him today,
March 15th, by finding three
valuable objects: one of gold, one
of silver, and one of bronze. "Bring
these to me in the Senate after
dark and I'll forget your debt," he
assures you. It is now noon...
```


8

WHERE DO I GO FROM HERE?

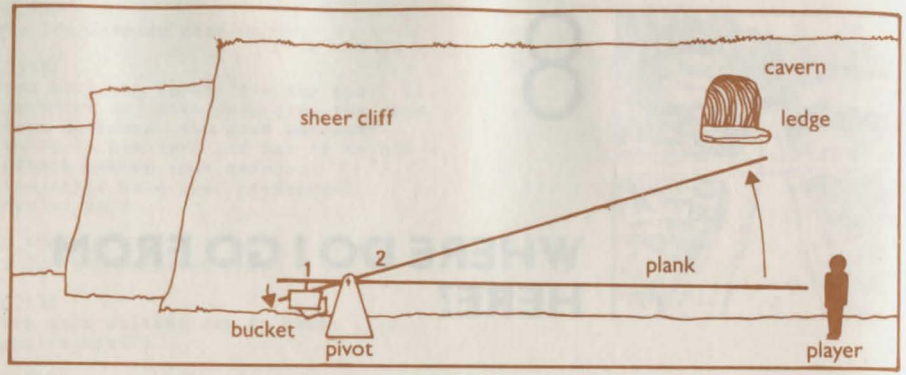
8.1 On plots and player enjoyment

In this book, we've paid a great deal of attention to the 'nuts and bolts' of Adventure games. In this last Part I want to forget about programming almost entirely and pay attention to what makes or breaks a game, namely the plot. A game can only become a 'classic' if it is fun to play, mystifying, and periodically gives the player that warm glow which comes from solving a well designed puzzle, without the aid of a crib sheet. The first rule to plot building is to forget almost entirely about game mechanics! Figure out the plot in words, not BASIC. Only when you've found something that holds together should you stop to ask if it's programmable.

The easiest way to explain how to go about creating one of those convoluted, 'multi-level' puzzle sequences is by giving an example of how it all begins.

I start with a pair of concepts and let them interact to form the foundation of a puzzle; although a plot theme can serve equally well. In this case, let's start with the first concept I ever thought up, but develop it differently. I remember programming it on my local mainframe over several lunch hours, purely for my own amusement. I went home on a Friday evening, having mentioned its existence to one person. On return to work Monday morning, my message space was full of notes from people I'd never heard of, relating bugs they'd found in my mini-game!

This concept is illustrated in Figs. 1 and 2. The idea was of an unclimbable cliff-face with an unreachable cavern, set high up and leading into the cliff. How was the player to reach it? He is provided with a long horizontal plank, several 'rooms' long, just off the ground and parallel to it. By 'room' I mean a describable area, not necessarily a real room. The plank is easily accessible along its entire length from the



ground, and can be walked upon. Near one end is a pivot fixed to the cliff, on which the plank can theoretically turn. On the other side of the pivot is a bucket with a fairly wide opening, but too narrow for the player to enter. Scattered around in fairly easy reach are a collection of extremely heavy objects, each as heavy as the player: an old lead bath, a locked coffin, and so on.

To reach the cavern, the player had to climb onto the pivot with one of the heavy objects and drop it in the bucket. As you can imagine, this weight causes the whole affair to tilt over, as in Fig. 2, with the plank now reaching seductively up to the cavern. All the player has to do – apparently – is to walk along and up the plank for several 'rooms' until the cavern entrance is attained.

That's the original concept in full. I wanted to create a problem where the geometry of the game could be drastically modified by the player's actions. Now, what can we do with the concept? Put another way, how can we make it more difficult for the player?

First of all, does it stand up to the laws of physics? Well, no, not really! Remember that the weight in the bucket is about the player's weight. Simple balances for levers will convince you that once the player has walked more than one 'room' along the tilted plank, his weight should cause the plank to tilt back downwards, throwing the player into the air with fatal consequences.

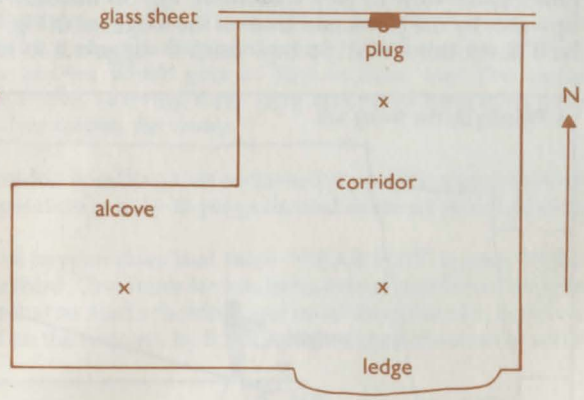
That's easy to handle. We allow the player to walk safely along the plank only the same number of steps as the number of heavy weights he has dropped in the bucket – so, in Fig. 2, three weights would be needed for the player to make it safely to the cavern. Obviously the player can only carry one of these weights at a time, which will make carting them around difficult; so a mental note is made not to leave them too far from the bucket (unless, of course, we can use one of them for another puzzle). Also, climbing the plank while carrying a heavy weight should be made fatal – the plank will break. Finally, let's make the plank break underneath the player just as he gets to the cavern, thus blocking his return. There will be another way out, even if we haven't thought of it yet, but creating a little honest worry in the player's mind never comes amiss!

On the face of it, we've exhausted the concept of the plank and pivot. It remains a single puzzle, certainly not one of those convoluted puzzles, which can take days to solve rather than minutes and usually involve interaction between more than one puzzle or concept.

Let's now consider what the player might find when he gets into the cavern. Obviously we must confront him with a problem whose solution will require the use of some object, objects, or information which he should have brought with him from below. Let's keep it simple. The cavern is a corridor leading into the cliff, ending in a huge sheet of glass, as in Fig. 3. (Ignore the alcove for a moment, we haven't thought of that yet). Beyond the glass, fish are swimming. (It was Alfred Hitchcock who said that for maximum enjoyment, an audience should be primed with all pertinent information before the denouement.) For the fun of it, let's put a large plug with a handle right in the middle of the glass. Given the facts, how is our player to get past the glass?

3 Plan of corridor

x Indicates an area, or 'room'



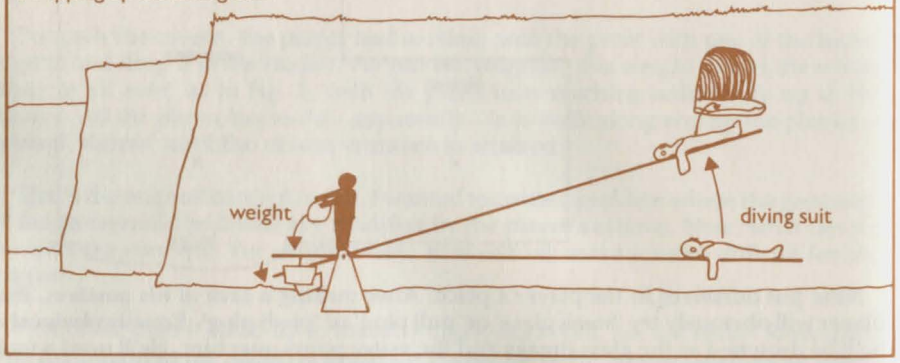
Now put ourselves in the player's place. After making a save of his position, the player will obviously try 'break glass' or 'pull plug' or 'push plug'. Equally obviously he'll be drowned as the glass breaks and the water pours over him. He'll need a tool for survival; so we give him an old diving suit. We can insist that the diving suit is actually worn to avoid drowning when he pulls the plug.

Yet, that seems too simple a solution. Surely the force of the water would swoosh him out through the corridor and over the cliff. Now that would be rather good. Our player finds out the hard way that breaking the glass drowns him; he realises that the old diving suit he saw earlier would be useful; carries it up the plank and wears it, tries again, and dies a different way! Excellent. If we can only make it more difficult to get the suit up to the cavern in the first place, he'll be even more certain he has the solution. (Have you noticed that we don't have a solution yet?) Our next problem is to stop him carrying the suit up the plank. How? A slippery patch between the second and third 'rooms' on the plank will do fine. When he is not carrying the suit, he can be warned that he nearly fell over because it was so slippery. Wearing the suit makes him so clumsy that he does fall off. What if he tries throwing the suit past the slippery patch? In that event, we must make it fall to the ground.

The astute player will wear his suit while climbing, thus avoiding the problem of clumsiness. We should applaud such astuteness by allowing him to get so far, but no further. If the suit is old, it won't have much oxygen left, each 'beat' that it is worn uses up one notch of oxygen. If we arrange the solution so that it requires all but one notch of the suit's oxygen, then walking up in the suit will use up a minimum of two notches (one to wear it, one to walk past the slippery area). The longer we can keep the player convinced that he has the solution, the more frustrated he will be when he finds out that he does not have it (and the more he'll appreciate his efforts when he finds the real solution).

So how does the player get the wretched suit up to the cavern, and what does he do with it there? Let's use the motion of the plank. If he leaves the suit on the end of the plank nearest the cavern and then goes and drops the weight into the bucket, it'll be waiting for him when he gets up there. But, no, that won't do, because we were going to have the plank break just as he reached the end, and we don't want to give him a game-turn to pick something up. So instead, we can have the suit thrown upwards by the plank and land on the ledge, as in Fig. 4. A fine solution, but indeed he'll never think of it! So how can we suggest it to him?

4 Flipping up the diving suit



This brings up the aspect of fairness to players. If a solution is in any way unlikely, or difficult to think of, then the player has to be given a clue. Now presumably he'll have messed around with that plank long before he gets to the stage of worrying about the suit. So when the plank tilts over, we can add a message about 'Particles of rubble on the end of the plank shoot up and land on the ledge', modifying the message if there were any objects on the end of the plank. This will plant a clue, but far, far before it can be used. (Another rule: give clues, yes, but it's quite fair to provide them well before they are useful!)

Now both the suit and the player are up in the corridor. How can the player break the glass and survive? If he could only be out of the way of the water's rush when it came, he could survive. So clearly he needs to be a little distance from the plug or glass when it breaks. Hence the corridor structure: if he can break the glass from one room away, he has every reason to hope for survival. He won't succeed, of course...

To pull the plug from some way away, we must supply a coil of rope, and let him 'TIE ROPE' to the plug. He can then back away, keeping hold of the rope (mental note: that'll be interesting to program!), put the suit on, and try 'PULL ROPE'. The plug will come off, breaking the glass, and the water will pour out, filling up the area in front of the glass. If our player stands there stupidly, the water gets him next turn instead of the one when the glass broke – but not quite with the same results.

There's another important point about puzzles. The player is now very near the solution, and it is vital that he be told this, or else he'll never find the real solution. So when the water hits him, he should get a very different message from what happened before. Something like "The force of the water has much decreased, but you're still swept out and over the cliff" will do. He has to realise that he's nearly correct, that's all.

His solution, given this knowledge, should now be obvious to him. He can't back away any further, because the geometry won't let him (i.e. the corridor hasn't any more rooms). But he can dodge sideways, into the alcove. So now he tries this, with 'PULL ROPE' ("The glass breaks, and water begins to pour...") followed by 'WEST', or whatever, to get into the alcove, which gets a "Just in time, too! The water swooshes past the end of the alcove, carrying rope, plug and everything in its path over the cliff. After a while, the waters die away."

Did you notice how we avoided handling that awkward rope with a plug tied on the end? This cleaning-up operation is easy to program and saves us much labour.

Now, how many notches of oxygen does that take? 'WEAR SUIT' is one; 'PULL ROPE' is another; 'WEST' is a third. One more for kindness is four notches of oxygen which we must supply – after that he starts choking, and must drop the suit. So as we surmised, if he wears the suit on the way up, he'll run out of oxygen too soon to solve the problem!

8.2 More on plots

There's a whole lot more we can do with this puzzle, both before and after the set-piece itself.

First of all, we can seek to use objects in unfamiliar ways. That avoids the dreadful setup typical of so many Adventure games in which the solution to every puzzle is 'do action A with object B'. At one level, this must always happen – you have to say something in the game to achieve your ends! But we can make it fun, too. Well, what objects have we so far? The two objects in the game (the suit and the rope) plus the heavy weights, have all been used 'normally': you would expect to wear a diving suit, tie a rope, and so on.

Can we add to the player's enjoyment by using some of these differently? For example, what of the heavy weights? Suppose we protected the diving suit against discovery by having a large scorpion run out and sting the player when he tried to take the suit? We could allow the player one game-turn to avoid the threat. This is rather a nice puzzle, as the obvious solution – put the suit on to protect your skin – (a) will not be allowed (the scorpion gets you anyway) and (b) wouldn't work as it would

waste too much oxygen. If the player tries throwing something at it, it can 'bounce off the scorpion's scaly hide'. This again is a non-null response from the game which signals to the player that he's on the right track; what he should have tried is throwing (or dropping, if we feel kind) one of the heavy objects. 'There is a sickening squelch, and a foul odour of squashed scorpion. . .'

On the same lines, can we re-use the suit? How about having a door some way beyond the glass area marked 'OGRE – DO NOT DISTURB', and a pair of glasses hanging high up by the door. The door is unopenable, but 'KNOCK' produces an irritable ogre who peers out short-sightedly, grabs his glasses and puts them on, mutters 'Oh, there you are!', and eats the player. The solution could be to jump up and grab the glasses (the noise from which brings the ogre anyway). But provided the player has left the suit standing up in the room, when the ogre enters, he can't find his glasses, so stares around until he sees the suit. Being short-sighted, he assumes it's the player and grabs it and eats it, while the player slips past him. Again, we must signal this to the player when he's near the solution. So if he's tried taking the glasses, but hasn't left the suit on the floor, we provide text like "The ogre peers around until he sees a man-sized object – you!" Or perhaps something a little less blatant; it depends how kind you're feeling!

This puzzle is an example of another key feature of Adventure game plots: interaction with other beings. Nothing is worse than a lifeless game in which the player meets only objects and words which he must manipulate to achieve ends. People make a game much more friendly.

Another, unfortunately rather rare Adventure problem is apparently no problem at all! Suppose, as is often the case, it was fatal to move without a light from one dark area to another. Further, suppose that the player was not allowed to carry anything else while carrying one of our heavy weights. We, the game's creators, then put a weight inside a dark region. The problem? to get the weight to the bucket? The poor player sees no problem at all, until he tries to do something. (The solution? Drop the lamp one move towards light from the heavy weight; move to weight, which is from a lit area and therefore safe; pick up weight; carry it past the lamp one move, which is also safe; drop the weight; get the lamp and piggy-back it past the weight one move; drop the lamp; etc., until daylight is reached.)

A similar example might concern a maze. Most mazes, as we have seen, are collections of contiguous areas, all of which share the same description. We all know the sinking feeling of 'You are in a collection of twisty little passages, all the same'! However, mazes can be made much more interesting, especially if the player is encouraged not to recognise the area as a maze. Suppose the maze was no more nor less than a room description which 'does' nothing at all. While manipulation of some object in the game becomes the cause a reaction which can enable the player to trace a tortuous path to reach his goal. Alternatively one could organize a what appears to be a maze solution, but is in fact a complete red herring. We have already met a maze which cannot be mapped the usual way – by dropping objects – because for some reason the objects won't stay put. Concocting such mazes is one of my delights in programming, I must confess.

One can also set explicit puzzles. For example, a piece of code written on a wall, which must be deciphered before the player can make use of the information it contains. Since not all players are expert cryptanalysts, a collection of equally cryptic hints must be left around to enable a player to decipher the code. Walls are very useful here; graffiti writers have a field day in Adventure games. By scattering the hints suitably, it may take the player a long time before he figures the solution out. In cases like that, placing the code near the player's starting area adds to the puzzlement.

Two other sources of puzzles are mathematics and thematic problems. The former can be disguised, and yet based firmly on some well-known mathematics problem, theorem, or whatever. With a little thought I suspect even Pythagoras' theorem could be made to serve for a puzzle. (Now there's an idea! Suppose our rope was five rooms long, however that's measured, but we have to pull it around two sides of a right-angled triangle, sides three and four rooms. Yes, there might be something there. . .) Thematic ideas often provide the smaller, 'filling-in' problems, essential to Adventures. After all, not every puzzle should be earth-shatteringly difficult, or our players will never get anywhere! So a few of the 'you can't do this until you've done this action' puzzles, which stand alone, are important, and can often be added while the main plot is being written down in legible form for the first time. That rope, for example – can we make it a little harder to get? It could be holding up a rotting corpse on a gallows, perhaps. But the stench of the corpse drives the player back. Could 'HOLD BREATH' be the solution? Or an approach from another direction to avoid the prevailing wind? The point is that (a) the problem stands alone, involving no other objects or problems and (b) it isn't too difficult.

APPENDICES

A1 What you need to know about bitwise logic

In the latter half of this book we have made frequent use of bitwise logical operations to insert, modify, or retrieve single zeros or ones from a byte holding eight properties. This technique uses what is known as 'bitwise logic'. This appendix very briefly discusses what is going on, but takes the subject only as far as we need.

When, in Part 2, we accessed individual digits in a decimal number, we had to use arithmetic operations to get at or change the digits. If we work to any base other than 10, we can still use arithmetic if we wish. Thus, to retrieve property 3 from a value X%, say, we can use (X% DIV 4) MOD 2. Here the 4 is 2 to the power (3 – 1). However, the beauty of binary storage – apart from the saving in space – is that there are neater ways of accessing the digits using bitwise logic.

You'll have used AND, OR, etc. in your other programming. They act on two logical expressions, each of which might be TRUE or FALSE, and produce the appropriate logical answer: AND yields TRUE only if both expressions are TRUE, OR gives TRUE if either expression is TRUE. You're less likely to have used them for comparing individual binary digits, although this is fully covered in the manuals.

Suppose we have the contents of two bytes, stored in X% and Y%. These, remember, are numbers, not logical expressions! What, then, is the result of X% AND Y%? Well, what happens is that BASIC writes out X% and Y% underneath each other, as if they were each a string of eight binary properties. For example, we might have:

```
X% 00101101
Y% 01001011
```

X% AND Y% is then evaluated by going through each vertical pair of digits and doing an AND on them. So the result for each pair is 0 unless both digits are 1's, giving X% AND Y% as

```
X% AND Y% 00001001
```

because only the 1's pairs yield 1's for answers. The answer is then treated as binary, giving 1 lot of 2 cubed plus 1 unit = 9 in base 10.

We can also try X% OR Y%, where each pair of digits gives 1 unless both are zero. Thus we have

```
X% OR Y% 01101111
```

which converts to 1 lot of 64 (2 to the 6th power) plus 1 lot of 32 plus 1 lot of 8 plus 1 lot of 4 plus 1 lot of 2 plus 1 = 111.

So how do we use these operations – which are vastly faster than arithmetic, hence their use – to access properties? Suppose first of all we want to retrieve property N, say, where N can be 0, 1, 2, . . . , or 7. Then if X% holds all the properties, we write

```
X% AND 2^N
```

which miraculously is nonzero if the Nth property is set, and 0 if it isn't! The reason can be found by a little experimentation. Let's write out X% symbolically like this – I've assumed N is 3 for definiteness:

```
PROPS 76543210
   X% ????P???
```

where '?' means a 1 or a zero, and we don't care which, and P means the property we're after. Now let's write 2N in the same form:

```
2^N 00001000
```

and now we AND this with X%. Each of the '?' digits is ANDded with a zero, so gives zero. The 'P' digit is ANDded with a 1. Now if P were 1, the answer would be 1; and if P were zero, the answer would be zero. Thus P AND 1 is P here, and the answer is:

```
X% AND 2 P0000P000
```

and this, as I claimed, is nonzero if P is 1 and zero if it isn't! As I noted earlier, you might want to make a table of values for 2N to speed things up a bit.

We can also use AND and OR to put specific digits into the set of properties. If we wish to put a 0 into digit N, we write

```
X% = X% AND (255 - 2^N)
```


which looks a little odd. Suppose again that N is 3. Then 255 – 2N = 255 – 8 = 243. This, written out in binary, is

```
11110111
```

which is all 1's except for the digit corresponding to property 3. ANDding this with X% will not alter any of the other digits (ANDding with 1 gives you what you had before, remember) but forces a zero into digit 3 (ANDding with zero forces zero). Thus we set property 3 to zero!

To set property N to 1 instead, we switch to ORring, and write

```
X% = X% OR 2^N
```

because 2^N in binary looks like:

```
00001000
```

with a 1 only at digit N. If we OR this with X%, all the zeros don't alter the appropriate digits of X% (ORring with zero gives you what you already had) and ORring with the 1 for digit N forces a 1 at that digit.

There's a lot more you can do with bitwise logic; but that suffices for our purposes.

A2 Hexadecimal notation

The other convenient piece of notation we use when creating Adventures is, occasionally, hexadecimal notation. Just as we've used numbers to base 10 in Part 2, and numbers to base 2 (or binary numbers) in Parts 4 to 7, there are occasional advantages to using other bases. The most useful of these is base 16 (or hexadecimal). There are three reasons for this. First, hexadecimal keeps the convenience of binary for logical operations, although we seldom need this. Second, the BBC BASIC interpreter is equally at home in hexadecimal or decimal notation, unlike most of us! Third, there are occasional things we do with the computer which require hexadecimal notation, and won't work in decimal.

BBC BASIC recognises numbers as hexadecimal if we begin them with an ampersand ('&'). Thus '15' is a decimal number, and means what it says. '&15', however, is a hexadecimal number because of the '&' at the beginning. You can write things like 'P% = &85' in BASIC and they'll be quite happily understood. Most people, myself included, can't look at a hexadecimal number and know how much it is in decimal, nor can they take a decimal number and immediately write it in hexadecimal. Acorn are obliging here. If one writes 'PRINT P%', say, out comes the value of P% in normal decimal. If one puts a tilde – " – before the expresssion to be printed, out it comes in hexadecimal (but without the '&', by the way). Thus converting from one to the other is easy. 'PRINT &54A6' will convert that hexadecimal expression to decimal; 'PRINT 81456' will convert that decimal expression to hexadecimal. End of conversion problem!

The actual notation looks nasty but isn't. Since we can have digits up to 15 – just as in decimal we could have digits up to 9, which is one less than 10, the base – we have to invent names for digits corresponding to 10, 11, 12, 13, 14 and 15. We use letters for these, with A standing for 10, B for 11, C for 12, D for 13, E for 14, and F for 15. The next number, 16, is '1 lot of 16 and zero units' and is thus &10. And so the counting goes. You can, of course, use the computer for any hexadecimal calculations you want to do; I do all the time.

Another advantage of hexadecimal is that bytes hold numbers from zero to 255. We can think of these as two-digit hexadecimal numbers, running from &00 to &FF – i.e. every possible combination in hexadecimal. Thus hexadecimal is in some sense more 'natural" than decimal for working with bytes. We shall use either, as it suits us. If you are even slightly unhappy, use decimal wherever you can manage it. Only the *LOAD and *SAVE commands used in the game positively require hexadecimal, and mere jottings on paper will solve that problem for you.

INTERESTED IN THE LATEST COMPUTER BOOKS AND SOFTWARE?

Penguin have many exciting future projects to share with you. There will be books on new models and machines, specific handbooks on graphics, sound and other functions, *plus* a terrific range of Penguin Software covering everything from arcade games to dieting!
We will keep you regularly in touch with the latest news. Just send your name, address and any special interests to:

Penguin Books Dept. CMD
536 Kings Road
London SW10 0UH

How To Write
ADVENTURE GAMES

This book is designed to teach readers who have started programming in BBC BASIC how to create and write fairly complicated adventure games, though the text is structured so that simple games can be written after reading only the first few parts.

Three games are created in the book: CAVES, a game of exploration through a random network of caves and passages in search of treasure and allies; MINI, a simple four-room adventure; and ROMAN, a complex adventure set in Ancient Rome.

Unlike other books on the subject, the reader is taken far beyond programming any specific adventure game. A multi-purpose 'shell' adventure program and a database creation program are provided for use when compiling any adventure. Several chapters are devoted entirely to plotting and puzzle creation, with the stress on new and different puzzle types.

How to Write Adventure Games is the complete book on the subject and particularly easy to follow.

PENGUIN ACORN COMPUTER LIBRARY

United Kingdom £5.95
AUST. $14.95 (recommended)
N.Z.$16.95

Computer Studies
ISBN 0 14
00 · 7814 2