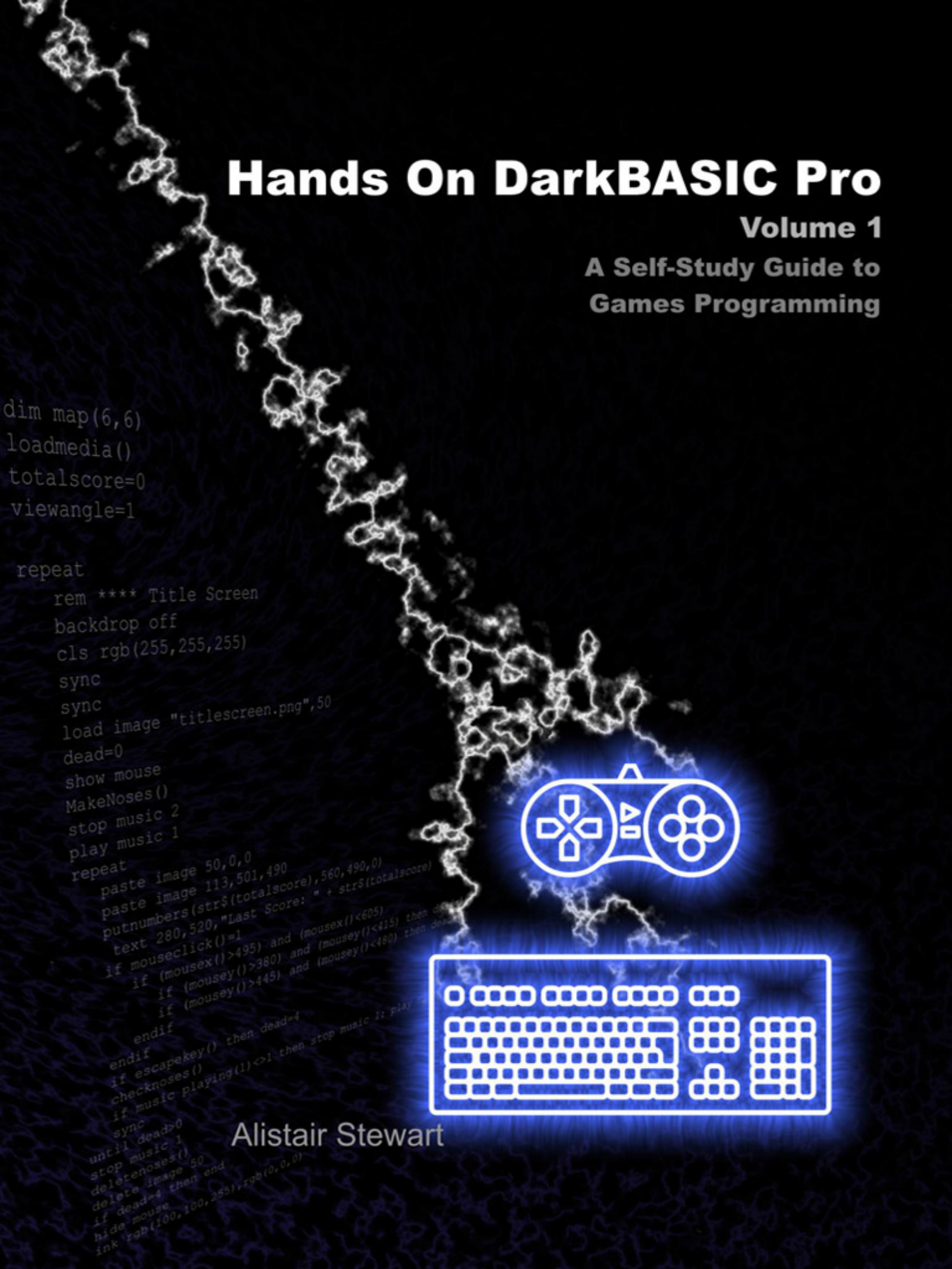
Hands On DarkBASIC Pro

Volume 1

A Self-Study Guide to Games Programming



Alistair Stewart

This book offers an easy-to-follow, self-study introduction to games software development using DarkBASIC Pro - a programming language designed specifically for creating video games on a PC.

If you are new to programming or just want to find out how to write computer games, this text will get you started.

Find out how to produce anything from a simple guessing game to a two-player, animated sprite space game. Use sound and video. Create games that use the mouse or a joystick.

Fundamental software design and implementation is also covered. Find out how to design and create modular programs, how to create test data and construct software using bottom-up and top-down methods. Use arrays, files, and record structures.

The hands-on approach used throughout the book means that most of your time is spent at the computer creating programs.

The numerous examples and exercises (with solutions included) lead you through both the basics and subtleties of the language. There are several complete games for you to study and modify.

Files used in the examples can be downloaded from our website.

In Volume 2

The second volume contains more advanced topics such as creating 3D graphics, textures, special effects, cameras, lights, internet access, and network programming.

Digital Skills

www.digital-skills.co.uk

ISBN 1-874107-08-4

9 781874 107088

Hands On DarkBASIC Pro

Volume 1

A Self-Study Guide to Games Programming

Alistair Stewart

Digital Skills
Milton
Barr
Girvan
Ayrshire
KA26 9TY

www.digital-skills.co.uk

DarkBASIC Professional is produced by The Game Creators Ltd

Cover Design by Neil King 2007

Printed September 2005
2nd Printing November 2005
3rd Printing January 2006
4th Printing November 2006
5th Printing February 2007
6th Printing September 2007
7th Printing February 2008
8th Printing September 2009 (with corrections)
9th Printing July 2010 (with corrections)

Title : Hands On DarkBASIC Pro Volume 1

ISBN : 1-874107-08-4
ISBN-13 978-1-874107-08-8

Other Titles Available:

Hands On DarkBASIC Pro Volume 2
Hands On Pascal
Hands On C++
Hands On Java
Hands On XHTML

Table Of Contents

Chapter 1 Designing Algorithms

Chapter 2 Starting DarkBASIC Pro

Chapter 3 Data

Chapter 5 Iteration

Chapter 6 Drawing Statements

Chapter 7 Modular Programming

Chapter 8 String Functions

Chapter 9 Hangman

Chapter 13 Bitmaps

Chapter 14 Video Cards and the Screen

Chapter 15 File Handling

Chapter 16 Handling Music Files

Chapter 17 Displaying Video Files

Chapter 20 Images

Chapter 21 Sprites1

Chapter 22 Sprites 2

Chapter 23 Animated Sprites

Chapter 24 Sound

Chapter 25 2D Vectors

Chapter 26 Space Duel

Chapter 27 Using the Mouse

Acknowledgements

I would like to thank all those who helped me prepare the final draft of this book.

In particular, Virginia Marshall who proof-read the original script and Michael Kerr who did an excellent job of checking the technical contents.

Any errors that remain are probably due to the extra few paragraphs I added after all the proof-reading was complete!

Thanks also to The Game Creators Ltd for producing an excellent piece of software - DarkBASIC Professional - known as DarkBASIC Pro to its friends.

Finally, thank you to every one of you who has bought this book. Any constructive comments would be most welcome.

Email me at *alistair@digital-skills.co.uk*.

Introduction

Welcome to a book that I hope is a little different from any other you've come across. Instead of just telling you about software design and programming, it makes you get involved. There's plenty of work for you to do since the book is full of exercises - most of them programming exercises - but you also get a full set of solutions, just in case you get stuck!

Learn by Doing

The only way to become a programming expert is to practice. No one ever learned any skill by just reading about it! Hence, this is not a text book where you can just sit back in a passive way and read from cover to cover whilst sitting in your favourite chair. Rather it is designed as a teaching package in which you will do most of the work.

The tasks embedded in the text are included to test your understanding of what has gone before and as a method of helping you retain the knowledge you have gained. It is therefore important that you tackle each task as you come to it. Also, many of the programming exercises are referred to, or expanded, in later pages so it is important that you are familar with the code concerned.

What You Need

You'll obviously need a PC and a copy of DarkBASIC Pro.

You don't need any experience of programming, but knowing your bits from your bytes and understanding binary and hexadecimal number systems would be useful.

How to Get the Most out of this Text

Experience has shown that readers derive most benefit from this material by approaching its study in an organised way. The following strategy for study is highly recommended:

1. Read a chapter or section through without taking notes or worrying too much about topics that are not immediately clear to you. This will give you an overview of the contents of that chapter/section.

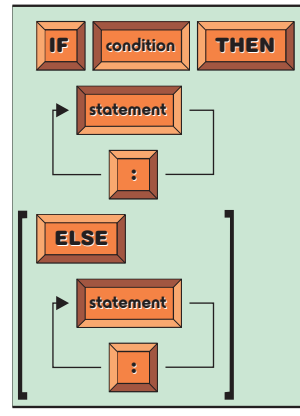
2. Re-read the chapter. This time take things slowly; make notes and summaries of the material you are reading (even if you understand the material, making notes helps to retain the facts in your long-term memory); re-read any parts you are unclear about.

3. Embedded in the material are a series of activities. Do each task as you reach it (on the second reading). These activities are designed to test your knowledge and understanding of what has gone before. Do not be tempted to skip over them, promise to come back to them later, or to make only a half-hearted attempt at tackling them before looking up the answer (there are solutions at the end of each chapter). Once you have attempted a task, look at the solution given. Often there will be important points emphasised in the solution which will aid higher understanding.

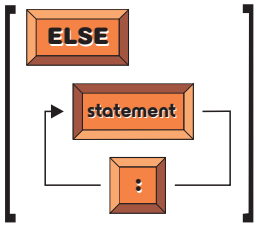
4. As you progress through the book, go back and re-read earlier chapters, since you will often get something new from them as your knowledge increases.

Language Syntax Diagrams

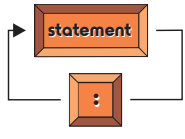
The text contains many syntax diagrams which give a visual representation of the format of various statements allowed in DarkBASIC Professional. These diagrams make no attempt to be complete, but merely act as a guide to the format most likely to be used. The accompanying text and example should highlight the more complex options available. Below is a typical diagram:



Each tile in the diagram holds a **token** of the statement. Raised tiles represent fixed terms in the statement, which must be entered exactly as shown. Sunken tiles represent tokens whose exact value is decided by you, the programmer, but again these values must conform to some stated rule.



Items enclosed in brackets may be omitted if not required. In this example we can see that ELSE and all the terms that follow may be omitted.



Where one or more tokens in a diagram may be repeated indefinitely, this is shown using the arrowed line. This example shows that any number of statements can be used so long as a colon appears between each statement.

Occasionally, a single line of code will have to be printed over two or more lines because of paper width restrictions; these lines are signified by a ↳ symbol. Enter these lines without a break when testing any of the programs in which they are used. For example, the code

```
SPRITE crosshairs,(JOYSTICK X()+1000)*xpixels#,
↳(JOYSTICK Y()+1000)*ypixels#,1
```

should be entered as a single line.

1

Designing Algorithms

Boolean expressions

Data Variables

Designing Algorithms

Desk Checking

IF Control Structure

FOR Control Structure

REPEAT Control Structure

Stepwise Refinement

Testing

WHILE Control Structure

Following Instructions

Activity 1.1

Carry out the following set of instructions in your head.

Think of a number between 1 and 10
Multiply that number by 9
Add up the individual digits of this new number
Subtract 5 from this total
Think of the letter at that position in the alphabet
Think of a country in Europe that starts with that letter
Think of a mammal that starts with the second letter of the country's name
Think of the colour of that mammal

Congratulations! You've just become a human computer. You were given a set of instructions which you have carried out (by the way, did you think of the colour grey?).

That's exactly what a computer does. You give it a set of instructions,the machine carries out those instructions, and that is ALL a computer does. If some computers seem to be able to do amazing things, that is only because someone has written an amazingly clever set of instructions. A set of instructions designed to perform some specific task is known as an **algorithm**.

There are a few points to note from the algorithm given above:

- There is one instruction per line
- Each instruction is unambiguous
- Each instruction is as short as possible

Activity 1.2

This time let's see if you can devise your own algorithm.

The task you need to solve is to measure out exactly 4 litres of water. You have two containers. Container A, if filled, will hold exactly 5 litres of water, while container B will hold 3 litres of water. You have an unlimited supply of water and a drain to get rid of any water you no longer need. It is not possible to know how much water is in a container if you only partly fill it from the supply.

If you managed to come up with a solution, see if you can find a second way of measuring out the 4 litres.

A B

As you can see, there are at least two ways to solve the problem given in Activity 1.2. Is one better than the other? Well, if we start by filling container A, the solution needs less instructions, so that might be a good guideline at this point when choosing which algorithm is best.

However, the algorithms that a computer carries out are not written in English like the instructions shown above, but in a more stylised form using a **computer programming language**. DarkBASIC Pro is one such language. The set of program language instructions which make up each algorithm is then known as a **computer program** or **software**.

Just as we may perform a great diversity of tasks by following different sets of instructions, so the computer can be made to carry out any task for which a program exists.

Computer programs are normally copied (or **loaded**) from a magnetic disk into the computer's memory and then executed (or **run**). Execution of a program involves the computer performing each instruction in the program one after the other. This it does at impressively high rates, possibly exceeding 2,000 million (or 2 billion) instructions per second (2,000 mips).

Depending on the program being run, the computer may act as a word processor, a database, a spreadsheet, a game, a musical instrument or one of many other possibilities. Of course, as a programmer, you are required to design and write computer programs rather than use them. And, more specifically, our programs in this text will be mainly games-related; an area of programming for which DarkBASIC Pro has been specifically designed.

Activity 1.3

1. A set of instructions that performs a specific task is known as what?

2. What term is used to describe a set of instructions used by a computer?

3. The speed of a computer is measured in what units?

Control Structures

Although writing algorithms and programming computers are certainly complicated tasks, there are only a few basic concepts and statements which you need to master before you are ready to start producing software. Luckily, the concepts are already familiar to you in everyday situations. If you examine any algorithm, no matter how complex, you will find it consists of three basic structures:

- **Sequence** where one statement follows on from another.
- **Selection** where a choice is made between two or more alternative actions.
- **Iteration** where one or more instructions are carried out over and over again.

These are explained in detail over the next few pages. All that is needed is to formalise the use of these structures within an algorithm. This formalisation better matches the structure of a computer program.

Sequence

A set of instructions designed to be carried out one after another, beginning at the first and continuing, without omitting any, until the final instruction is completed,

is known as a **sequence**. For example, instructions on how to play Monopoly might begin with the sequence:

```
Choose your playing piece
Place your piece on the GO square
Get £1,500 from the bank
```

The set of instructions given earlier in Activity 1.1 is also an example of a sequence.

Activity 1.4

Re-arrange the following instructions to describe how to play a single shot during a golf game:

```
Swing club forwards, attempting to hit ball
Take up correct stance beside ball
Grip club correctly
Swing club backwards
Choose club
```

Selection

Binary Selection

Often a group of instructions in an algorithm should only be carried out when certain circumstances arise. For example, if we were playing a simple game with a young child in which we hide a sweet in one hand and allow the child to have the sweet if she can guess which hand the sweet is in, then we might explain the core idea with an instruction such as

```
Give the sweet to the child if the child guesses which hand the sweet is in
```

Notice that when we write a sentence containing the word IF, it consists of two main components:

a condition : *the child guesses which hand the sweet is in*

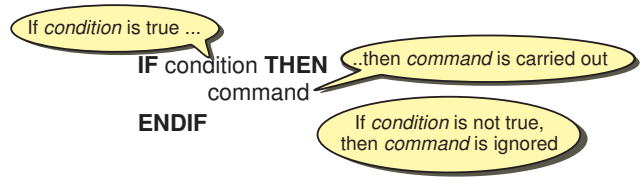
and

a command : *give the sweet to the child*

A **condition** (also known as a **Boolean expression**) is a statement that is either true or false. The command given in the statement is only carried out if the condition is true and hence this type of instruction is known as an **IF** statement and the command as a **conditional instruction**. Although we could rewrite the above instruction in many different ways, when we produce a set of instructions in a formal manner, as we are required to do when writing algorithms, then we use a specific layout as shown in FIG-1.1 always beginning with the word IF.

FIG-1.1

The IF Statement



Notice that the layout of this instruction makes use of three terms that are always included. These are the words IF, which marks the beginning of the instruction; THEN, which separates the condition from the command; and finally, ENDIF which marks the end of the instruction.

The indentation of the command is important since it helps our eye grasp the structure of our instructions. Appropriate indentation is particularly valuable in aiding readability once an algorithm becomes long and complex. Using this layout, the instruction for our game with the child would be written as:

```
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ENDIF
```

Sometimes, there will be several commands to be carried out when the condition specified is met. For example, in the game of Scrabble we might describe a turn as:

```
IF you can make a word THEN
    Add the word to the board
    Work out the points gained
    Add the points to your total
    Select more letter tiles
ENDIF
```

Of course, the conditional statement will almost certainly appear in a longer sequence of instructions. For example, the instructions for playing our guessing game with the young child may be given as:

```
Hide a sweet in one hand
Ask the child to guess which hand contains the sweet
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ENDIF
Ask the child if they would like to play again
```

This longer sequence of instructions highlights the usefulness of the term ENDIF in separating the conditional command, Give the sweet to the child, from subsequent unconditional instructions, in this case, Ask the child if they would like to play again.

Activity 1.5

A simple game involves two players. Player 1 thinks of a number between 1 and 100, then Player 2 makes a single attempt at guessing the number. Player 1 responds to a correct guess by saying *Correct*. The game is then complete and Player 1 states the value of the number.

Write the set of instructions necessary to play the game.

In your solution, include the statements:

```
Player 1 says "Correct"
Player 1 thinks of a number
IF guess matches number THEN
```

The IF structure is also used in an extended form to offer a choice between two alternative actions. This expanded form of the IF statement includes another formal term, ELSE, and a second command. If the condition specified in the IF statement is true, then the command following the term THEN is executed, otherwise that following ELSE is carried out.

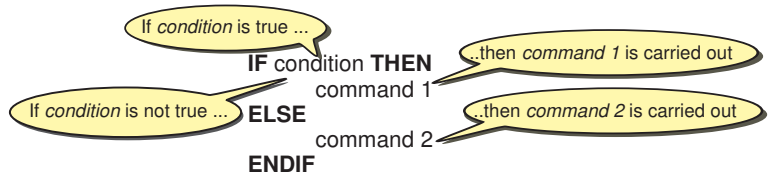
For instance, in our earlier example of playing a guessing game with a child, nothing happened if the child guessed wrongly. If the person holding the sweet were to eat it when the child's guess was incorrect, we could describe this setup with the following statement:

```
IF the child guesses which hand the sweet is in THEN
      Give the sweet to the child
ELSE
      Eat sweet yourself
ENDIF
```

The general form of this extended IF statement is shown in FIG-1.2.

FIG-1.2

The IF ... ELSE Statement



Activity 1.6

Write an IF statement containing an ELSE section which describes the alternative actions to be taken when playing Hangman and the player trying to guess the word suggests a letter.

In the solution include the statements:

```
Add letter at appropriate position(s)
Add part to hanged man
```

Choosing between two alternative actions is called **binary selection**.

When we have several independent selections to make, then we may use several IF statements. For example, when playing Monopoly, we may buy any unpurchased property we land on. In addition, we get another turn if we throw a double. This part of the game might be described using the following statements:

```
Throw the dice
Move your piece forward by the number indicated
IF you land on an unpurchased property THEN
      Buy the property
ENDIF
IF you threw doubles THEN
      Throw the dice again
ELSE
      Hand the dice to the next player
ENDIF
```

This set of instructions is not complete and is shown here only to illustrate the use of multiple IF statements in an algorithm.

Multi-way Selection

Although a single IF statement can be used to select one of two alternative actions, sometimes we need to choose between more than two alternatives (known as **multi-way selection**). For example, imagine that the rules of the simple guessing game mentioned in Activity 1.5 are changed so that there are three possible responses to Player 2's guess; these being:

- Correct
- Too low
- Too high

One way to create an algorithm that describes this situation is just to employ three separate IF statements:

```
IF the guess is equal to the number you thought of THEN
    Say "Correct"
ENDIF
IF the guess is lower than the number you thought of THEN
    Say "Too low"
ENDIF
IF the guess is higher than the number you thought of THEN
    Say "Too high"
ENDIF
```

This will work, but would not be considered a good design for an algorithm since, when the first IF statement is true, we still go on and check if the conditions in the second and third IF statements are true. After all, only one of the three conditions can be true at any one time.

Where only one of the conditions being considered can be true at a given moment in time, these conditions are known as **mutually exclusive** conditions.

The most effective way to deal with mutually exclusive conditions is to check for one condition, and only if this is not true, are the other conditions tested. So, for example, in our algorithm for guessing the number, we might begin by writing:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    ***Check the other conditions***
ENDIF
```

Of course a statement like ***Check the other conditions*** is too vague to be much use in an algorithm (hence the asterisks). But what are these other conditions? They are *the guess is lower than the number you thought of* and *the guess is higher than the number you thought of*.

We already know how to handle a situation where there are only two alternatives: use an IF statement. So we can chose between *Too low* and *Too high* with the statement

```
IF guess is less than number THEN
    Say "Too low"
ELSE
    Say "Too high"
ENDIF
```

Now, by replacing the phrase ***Check the other conditions*** in our original algorithm with our new IF statement we get:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    IF guess is less than number THEN
        Say "Too low"
    ELSE
        Say "Too high"
    ENDIF
ENDIF
```

Notice that the second IF statement is now totally contained within the ELSE section of the first IF statement. This situation is known as **nested** IF statements. Where there are even more mutually exclusive alternatives, several IF statements may be nested in this way. However, in most cases, we're not likely to need more than two nested IF statements.

Activity 1.7

In an old TV programme called The Golden Shot, contestants had to direct a crossbow in order to shoot an apple. The player sat at home and directed the crossbow controller via the phone. Directions were limited to the following phrases: *up a bit*, *down a bit*, *left a bit*, *right a bit*, and *fire*.

Write a set of nested IF statements that determine which of the above statements should be issued. Use statements such as:

```
IF the crossbow is pointing too high THEN
```

and

```
Say "Left a bit"
```

As you can see from the solution to Activity 1.7, although nested IF statements get the job done, the general structure can be rather difficult to follow. A better method would be to change the format of the IF statement so that several, mutually exclusive, conditions can be declared in a single IF statement along with the action required for each of these conditions. This would allow us to rewrite the solution to Activity 1.7 as:

```
IF
    crossbow is too high:
        Say "Down a bit"
    crossbow is too low:
        Say "Up a bit"
    crossbow is too far right:
        Say  "Left a bit"
    crossbow is too far left:
        Say  " Right a bit"
    crossbow is on target:
        Say "Fire"
ENDIF
```

Each option is explicitly named (ending with a colon) and only the one which is true will be carried out, the others will be ignored.

Of course, we are not limited to merely five options; there can be as many as the situation requires.

When producing a program for a computer, all possibilities have to be taken into account. Early adventure games, which were text based, allowed the player to type a command such as *Go East*, *Go West*, *Go North*, *Go South* and this moved the player's character to new positions in the imaginary world of the computer program. If the player typed in an unrecognised command such as *Go North-East* or *Move faster*, then the game would issue an error message. This setup can be described by adding an ELSE section to the structure as shown below:

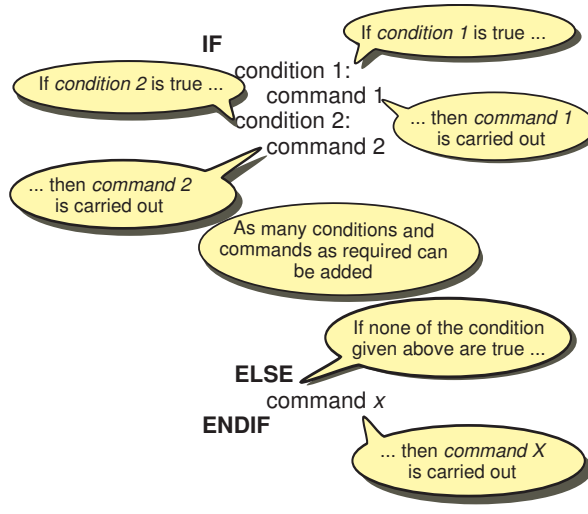
```
IF
    command is Go East:
        Move player's character eastward
    command is Go West:
        Move player's character westward
    command is Go North:
        Move player's character northward
    command is Go South:
        Move player's character southward
    ELSE
        Display an error message
ENDIF
```

The additional ELSE option will be chosen only if none of the other options are applicable. In other words, it acts like a catch-all, handling all the possibilities not explicitly mentioned in the earlier conditions.

This gives us the final form of this style of the IF statement as shown in FIG-1.3:

FIG-1.3

The Third Version of the IF Statement



Activity 1.8

In the TV game Wheel of Fortune (where you have to guess a well-known phrase), you can, on your turn, either guess a consonant, buy a vowel, or make a guess at the whole phrase.

If you know the phrase, you should make a guess at what it is; if there are still many unseen letters, you should guess a consonant; as a last resort you can buy a vowel.

Write an IF statement in the style given above describing how to choose from the three options.

Complex Conditions

Often the condition given in an IF statement may be a complex one. For example, in the TV game Family Fortunes, you only win the star prize if you get 200 points and guess the most popular answers to a series of questions. This can be described in our more formal style as:

```
IF at least 200 points gained AND all most popular answers have been guessed THEN
      winning team get the star prize
ENDIF
```

The AND Operator

Note the use of the word AND in the above example. AND (called a **Boolean operator**) is one of the terms used to link simple conditions in order to produce a more complex one (known as a **complex condition**). The conditions on either side of the AND are called the operands. Both operands must be true for the overall result to be true. We can generalise this to describe the AND operator as being used in the form:

```
condition 1  AND  condition 2
```

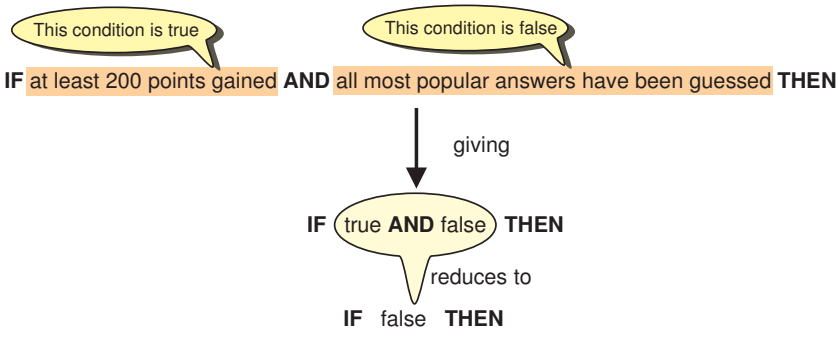
The result of the AND operator is determined using the following rules:

1. Determine the truth of condition 1
2. Determine the truth of condition 2
3. IF both conditions are true THEN
   the overall result is true
   ELSE
   the overall result is false
   ENDIF

For example, if we assume the group reaching the final of the game show Family Fortunes has amassed 230 points but have not guessed all of the most popular answers, then a computer would determine the overall result of the IF statement given earlier as shown in FIG-1.4.

FIG-1.4

Calculating the Result of an AND Operation



With two conditions there are four possible combinations. The first possibility is that both conditions are false; another possibility is that *condition 1* is false but *condition 2* is true.

> **Activity 1.9**
>
> What are the other two possible combinations of true and false?

The results of the AND operator are summarised in TABLE-1.1.

TABLE-1.1

The AND Operator

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

> **Activity 1.10**
>
> In the card game Snap, you win the cards on the table if you are first to place your hand over those cards, and the last two cards laid down are of the same value.
>
> Write an IF statement, which includes the term AND, summarising this situation.

The OR Operator

Simple conditions may also be linked by the Boolean OR operator. Using OR, only one of the conditions needs to be true in order to carry out the action that follows. For example, in the game of Monopoly you go to jail if you land on the *GoTo Jail*

square or if you throw three doubles in a row. This can be written as:

```
IF player lands on Go To Jail OR player has thrown 3 pairs in a row THEN
      Player goes to jail
ENDIF
```

Like AND, the OR operator works on two operands:

```
condition 1   OR   condition 2
```

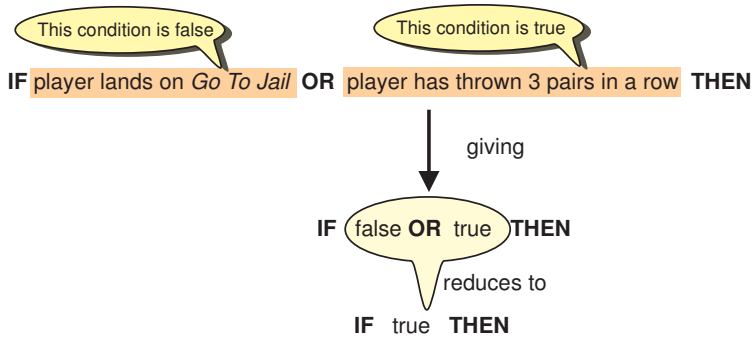
When OR is used, only one of the conditions involved needs to be true for the overall result to be true. Hence the results are determined by the following rules:

```
1.  Determine the truth of condition 1
2.  Determine the truth of condition 2
3.  IF any of the conditions are true THEN
        the overall result is true
    ELSE
        the overall result is false
    ENDIF
```

For example, if a player in the game of Monopoly has not landed on the *Go To Jail* square, but has thrown three consecutive pairs, then the result of the IF statement given above would be determined as shown in FIG-1.5.

FIG-1.5

Calculating the Result of an OR Operation



The results of the OR operator are summarised in TABLE-1.2.

TABLE-1.2

The OR Operator

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

Activity 1.11

In Monopoly, a player can get out of jail if he throws a double or pays a £50 fine.

Express this information in an IF statement which makes use of the OR operator.

The NOT Operator

The final Boolean operator which can be used as part of a condition is NOT. This operator is used to reverse the meaning of a condition. Hence, if *property mortgaged* is true, then *NOT property mortgaged* is false.

Notice that the word NOT is always placed at the start of the condition and not where it would appear in everyday English (*property NOT mortgaged*).

In Monopoly a player can charge rent on a property as long as that property is not mortgaged. This situation can be described with the statement:

```
IF NOT property mortgaged THEN
      Rent can be charged
ENDIF
```

The NOT operator works on a single operand:

```
NOT condition
```

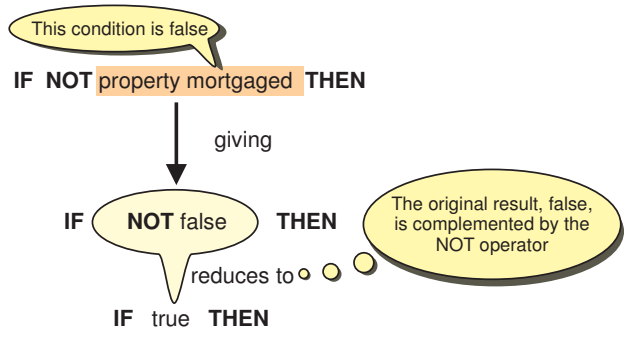
When NOT is used, the result given by the original condition is reversed. Hence the results are determined by the following rules:

1. Determine the truth of the condition
2. Complement the result obtained in step 1

For example, if a player lands on a property that is not mortgaged, then the result of the IF statement given above would be determined as shown in FIG-1.6.

FIG-1.6

Calculating the Result of a NOT Operation



The results of the NOT operator are summarised in TABLE-1.3.

TABLE-1.3

The NOT Operator

condition	NOT condition
false	true
true	false

Complex conditions are not limited to a single occurrence of a Boolean operator, hence it is valid to have statements such as:

```
IF player lands on Go To Jail OR player has thrown 3 pairs in a row OR
  player lifts a Go To Jail card
THEN
      Player goes to jail
ENDIF
```

Although us humans might be able to work all of this out in our heads without even a conscious thought, computers deal with such complex conditions in a slow, but methodical way.

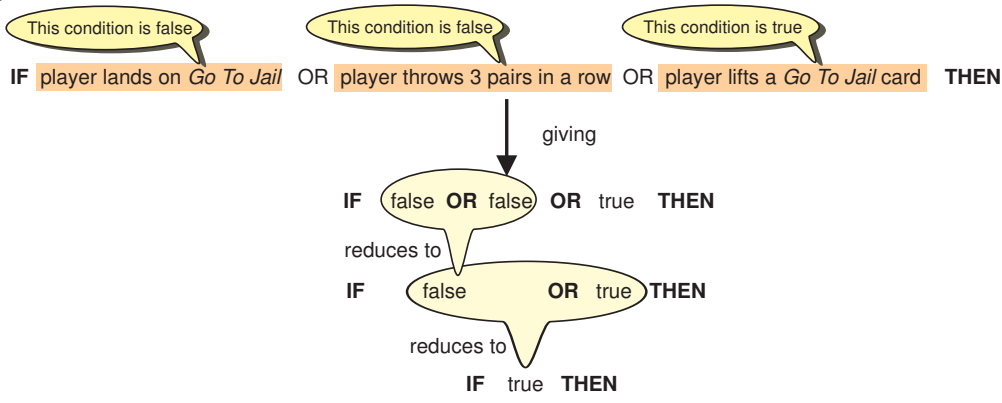
To calculate the final result of the condition given above, the computer requires several operations to be performed. These are performed in two stages:

1. Determine the truth of each condition
2. Determine the result of each OR operation, starting with the left-most OR

FIG-1.7

Using More than One OR Operator

For example, if a player lifts a *Go To Jail* card from the *Chance* pack, then the result of the IF statement given above would be determined as shown in FIG-1.7.



That might seem a rather complicated way of achieving what was probably an obvious result, but when the conditions become even more complex, this methodical approach is necessary.

Notice that when a complex condition contains only a single Boolean operator type (OR in the example above), that the expression is worked out from left to right. However, should the condition contain a mixture of OR, AND and NOT operators, NOT operations are performed first, ANDs second, and ORs last.

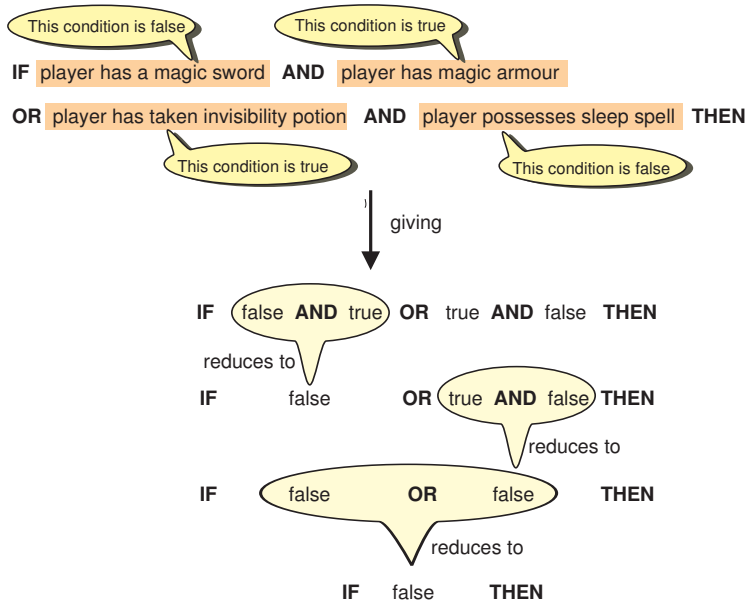
For example, if a game has the following rule

```
IF player has a magic sword AND player has magic armour OR
player has taken invisibility potion AND player possesses sleep spell
THEN
      Player can kill dragon
ENDIF
```

and a player has magic armour and has drunk the invisibility potion, then to determine if the player can kill the dragon, the process shown in FIG-1.8 is followed.

FIG-1.8

AND Operators have Priority



The final result shows that the player cannot kill the dragon.

Activity 1.12

A game has the following rule:

```
IF a player has an Ace AND player has King OR player has two Knaves THEN
    Player must pick up extra card
ENDIF
```

Using a similar approach to that shown in FIG-1.8 above, show the steps involved in deciding if the player should take an extra card assuming the player already has an Ace and one Knave.

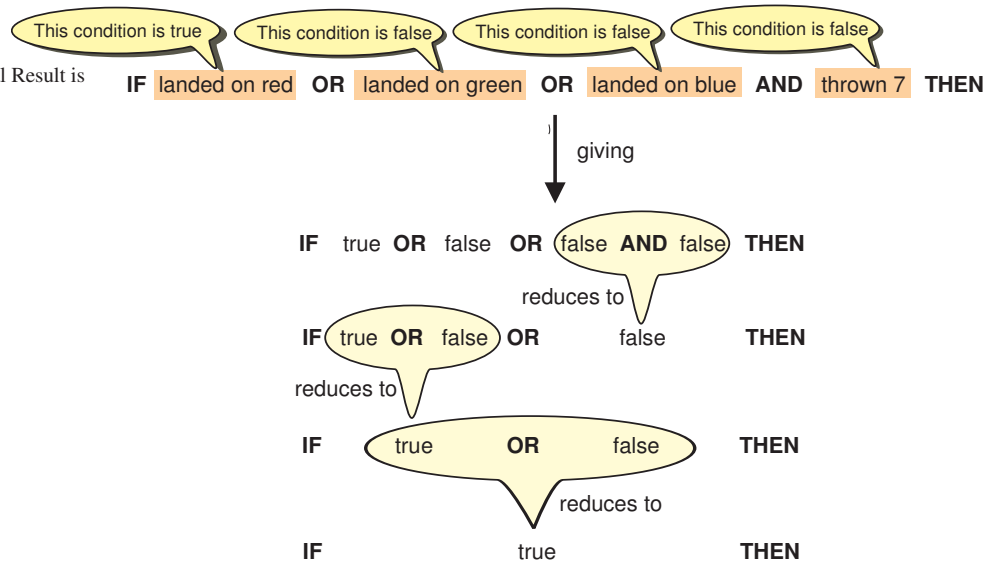
Sometimes the priority of operators works against what we are trying to express. For example, if a player receives a bonus if he lands on a red, green or blue square after throwing 7 on a pair of dice, then we might be tempted to write:

```
IF landed on red OR landed on green OR landed on blue AND thrown 7 THEN
    Add bonus to player's score
ENDIF
```

We would not expect a player landing on a red square after throwing 9 to receive the bonus. But, if we look at the calculation for such a situation, we get the result shown in FIG-1.9 which means that the bonus is incorrectly added to the player's score.

FIG-1.9

How the Final Result is Calculated



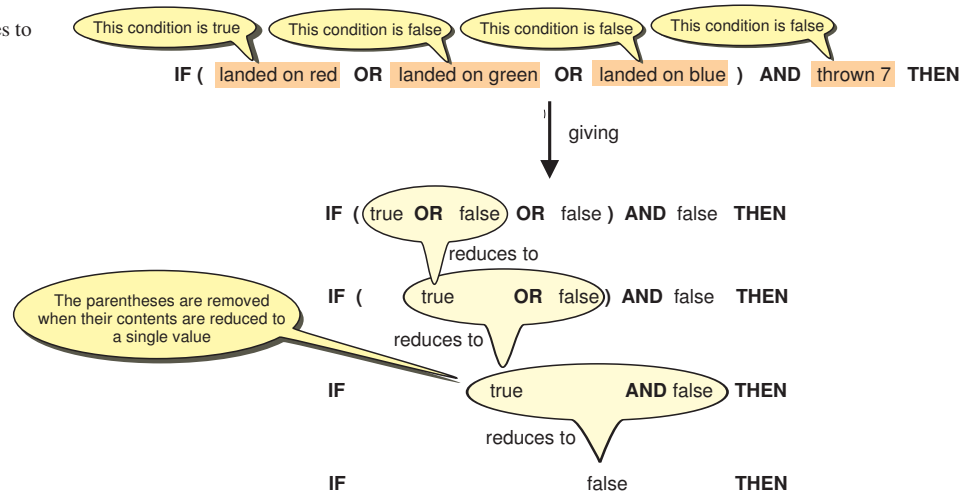
To achieve the correct results, we need the OR operations to be performed first and this can be done by giving the OR operators a higher priority than the AND. Luckily, operator priority can be modified by using parentheses. Operations in parentheses are always performed first. So, by rewriting our instruction as

```
IF (landed on red OR landed on green OR landed on blue) AND thrown 7 THEN
    Add bonus to player's score
ENDIF
```

the condition is calculated as shown in FIG-1.10.

FIG-1.10

Using Parentheses to Modify Operator Priority



Boolean operator priority is summarised in TABLE-1.4.

TABLE-1.4

Operator Priority

Priority	Operator
1	( )
2	NOT
3	AND
4	OR

Activity 1.13

The rules for winning a card game are that your hand of 5 cards must add up to exactly 43 (faces =10, Ace = 11) or you must have four cards of the same value. In addition, a player cannot win unless he has a Queen in his hand.

Express these winning conditions as an IF statement.

Activity 1.14

1. Name the three types of control structures.
2. Another term for condition is what?
3. Name the two types of selection.
4. What does the term *mutually exclusive conditions* mean?
5. Give an example of a Boolean operator.
6. If the terms AND and OR are included in a single complex condition, which of these operators will be performed first?
7. How can the order in which operations in a complex condition be changed?

Iteration

There are certain circumstances in which it is necessary to perform the same sequence of instructions several times. For example, let's assume that a game involves throwing a dice three times and adding up the total of the values thrown. We could write instructions for such a game as follows:

```
Set the total to zero
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Call out the value of total
```

You can see from the above that two instructions,

```
Throw dice
Add dice value to total
```

are carried out three times, once for each turn taken by the player. Not only does it seem rather time-consuming to have to write the same pair of instructions three times, but it would be even worse if the player had to throw the dice 10 times!

What is required is a way of showing that a section of the instructions is to be repeated a fixed number of times. Carrying out one or more statements over and over again is known as **looping** or **iteration**. The statement or statements that we want to perform over and over again are known as the **loop body**.

Activity 1.15

What statements make up the loop body in our dice problem given above?

FOR..ENDFOR

When writing a formal algorithm in which we wish to repeat a set of statements a specific number of times, we use a **FOR..ENDFOR** structure.

There are two parts to this statement. The first of these is placed just before the loop body and in it we state how often we want the statements in the loop body to be carried out. For the dice problem our statement would be:

```
FOR 3 times DO
```

Generalising, we can say this statement takes the form

```
FOR value times DO
```

where *value* would be some positive number.

Next come the statements that make up the loop body. These are indented:

```
FOR 3 times DO
    Throw dice
    Add dice value to total
```

Finally, to mark the fact that we have reached the end of the loop body statements we add the word ENDFOR:

Note that ENDFOR is left-aligned with the opening FOR statement.

```
FOR 3 times DO
      Throw dice
      Add dice value to total
ENDFOR
```

Now we can rewrite our original algorithm as:

```
Set the total to zero
FOR 3 times DO
      Throw dice
      Add dice value to total
ENDFOR
Call out the value of total
```

The instructions between the terms FOR and ENDFOR are now carried out three times.

Activity 1.16

If the player was required to throw the dice 10 times rather than 3, what changes would we need to make to the algorithm?
If the player was required to call out the average of these 10 numbers, rather than the total, show what other changes are required to the set of instructions.

You can find the average of the 10 numbers by dividing the final total by 10.

We are free to place any statements we wish within the loop body. For example, the last version of our number guessing game produced the following algorithm

```
Player 1 thinks of a number between 1 and 100
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
      Player 1 says "Correct"
ELSE
      IF guess is less than number THEN
            Player 1 says "Too low"
      ELSE
            Player 1 says "Too high"
      ENDIF
ENDIF
```

player 2 would have more chance of winning if he were allowed several chances at guessing player 1's number. To allow several attempts at guessing the number, some of the statements given above would have to be repeated.

Activity 1.17

What statements in the algorithm above need to be repeated?

To allow for 7 attempts our new algorithm becomes:

```
Player 1 thinks of a number between 1 and 100
FOR 7 times DO
      Player 2 makes an attempt at guessing the number
      IF guess matches number THEN
            Player 1 says "Correct"
      ELSE
            IF guess is less than number THEN
                  Player 1 says "Too low"
            ELSE
                  Player 1 says "Too high"
            ENDIF
      ENDIF
ENDFOR
```

Activity 1.18

Can you see a practical problem with the algorithm?
If not, try playing the game a few times, playing exactly according to the instructions in the algorithm.

Activity 1.19

During a lottery draw, two actions are performed exactly 6 times. These are:

```
Pick out ball
Call out number on the ball
```

Add a FOR loop to the above statements to create an algorithm for the lottery draw process.

Occasionally, we may have to use a slightly different version of the FOR loop. Imagine we are trying to write an algorithm explaining how to decide who goes first in a game. In this game every player throws a dice and the player who throws the highest value goes first. To describe this activity we know that each player does the following task:

```
Player throws dice
```

But since we can't know in advance how many players there will be, we write the algorithm using the statement

```
FOR every player DO
```

to give the following algorithm

```
FOR every player DO
    Throw dice
ENDFOR
Player with highest throw goes first
```

If we had to save the details of a game of chess with the intention of going back to the game later, we might write:

```
FOR each piece on the board DO
    Write down the name and position of the piece
ENDFOR
```

Activity 1.20

A game uses cards with images of warriors. At one point in the game the player has to remove from his hand every card with an image of a knight. To do this the player must look through every card and, if it is a knight, remove the card.

Write down a set of instructions which performs the task described above.

Your solution should include the statements

```
FOR every card in player's hand DO
```

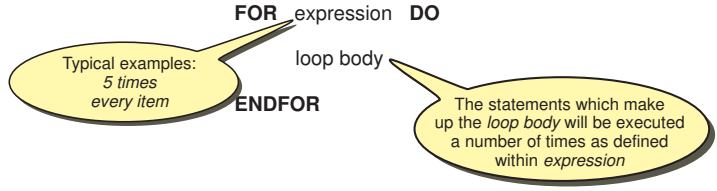
and

```
IF card is a knight THEN
```

The general form of the FOR statement is shown in FIG-1.11.

FIG-1.11

The FOR Loop



Although the FOR loop allows us to perform a set of statements a specific number of times, this statement is not always suitable for the problem we are trying to solve. For example, in the guessing game we stated that the loop body was to be performed 7 times, but what if player 2 guesses the number after only three attempts? If we were to follow the algorithm exactly (as a computer would), then we must make four more guesses at the number even after we know the correct answer!

To solve this problem, we need another way of expressing looping which does not commit us to a specific number of iterations.

REPEAT.. UNTIL

The REPEAT .. UNTIL statement allows us to specify that a set of statements should be repeated until some condition becomes true, at which point iteration should cease. The word REPEAT is placed at the start of the loop body and, at its end, we add the UNTIL statement. The UNTIL statement also contains a condition, which, when true, causes iteration to stop. This is known as the **terminating** (or **exit**) **condition**. For example, we could use the REPEAT.. UNTIL structure rather than the FOR loop in our guessing game algorithm. The new version would then be:

```
Player 1 thinks of a number between 1 and 100
REPEAT
      Player 2 makes an attempt at guessing the number
      IF guess matches number THEN
            Player 1 says "Correct
      ELSE
            IF guess is less than number THEN
                  Player 1 says "Too low"
            ELSE
                  Player 1 says "Too high"
            ENDIF
      ENDIF
UNTIL player 2 guesses correctly
```

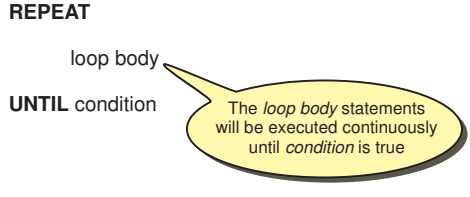
We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-armed bandit) is played:

```
REPEAT
      Put coin in machine
      Pull handle
      IF you win THEN
            Collect winnings
      ENDIF
UNTIL you want to stop
```

The general form of this structure is shown in FIG-1.12.

FIG-1.12

The REPEAT Loop



The terminating condition may use the Boolean operators AND, OR and NOT as well as parentheses, where necessary.

Activity 1.21

A one-armed bandit costs 50p per play. A player has several 50p pieces and is determined to play until his coins are gone or until he wins at least £10.00. Write an algorithm describing the steps in this game. The algorithm should make use of the following statements:

```
Collect winnings
Place coin in machine
Pull arm
UNTIL all coins are gone OR winnings are at least £10.00
```

There is still a problem with our number-guessing game. By using a REPEAT .. UNTIL loop we are allowing player 2 to have as many guesses as needed to determine the correct number. That doesn't lead to a very interesting game. Later we'll discover how we might solve this problem.

WHILE.. ENDWHILE

A final method of iteration, differing only subtly from the REPEAT.. UNTIL loop, is the WHILE .. ENDWHILE structure which has an **entry condition** at the start of the loop.

The aim of the card game of Pontoon is to attempt to make the value of your cards add up to 21 without going over that value. Each player is dealt two cards initially but can repeatedly ask for more cards by saying "twist". One player is designated the dealer. The dealer must twist while his cards have a total value of less than 16. So we might write the rules for the dealer as:

```
Calculate the sum of the initial two cards
REPEAT
      Take another card
      Add new card's value to sum
UNTIL sum is greater than or equal to 16
```

But this solution implies that the dealer must take at least one card before deciding to stop. Using the WHILE..ENDWHILE structure we could describe the logic as

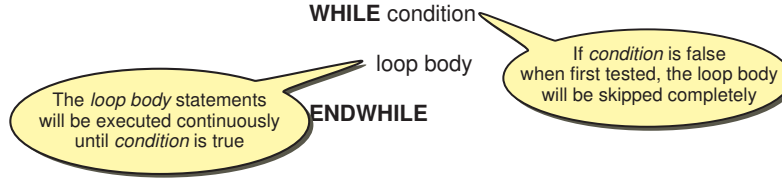
```
Calculate sum of the initial two cards
WHILE sum is less than 16 DO
      Take another card
      Add new card's value to sum
ENDWHILE
```

Now determining if the sum is less than 16 is performed before *Take another card* instruction. If the dealer's two cards already add up to 16 or more, then the *Take another card* instruction will be ignored.

The general form of the WHILE.. ENDWHILE statement is shown in FIG-1.13.

FIG-1.13

The WHILE Loop



In what way does this differ from the REPEAT statement? There are two differences:

- The condition is given at the beginning of the loop.
- Looping stops when the condition is false.

The main consequence of this is that it is possible to bypass the loop body of a WHILE structure entirely without ever carrying out any of the instructions it contains, whereas the loop body of a REPEAT structure will always be executed at least once.

Activity 1.22

A game involves throwing two dice. If the two values thrown are not the same, then the dice showing the lower value must be rolled again. This process is continued until both dice show the same value.

Write a set of instructions to perform this game.

Your solution should contain the statements

Roll both dice

and

Choose dice with lower value

Activity 1.23

1. What is the meaning of the term iteration?
2. Name the three types of looping structures.
3. What type of loop structure should be used when looping needs to occur an exact number of times?
4. What type of loop structure can bypass its loop body without ever executing it?
5. What type of loop contains an exit condition?

Data

Almost every game requires the players to remember or record some facts and figures. In our number guessing game described earlier, the players needed to remember the original number and the guesses made; in Hangman the word being guessed and the letters guessed so far must be remembered.

These examples introduce the need to process facts and figures (known as **data**). Every computer game has to process data. This data may be the name of a character, the speed of a missile, the strength of a blow, or some other factor.

Every item of data has two basic characteristics :

a name
and a value

The name of a data item is a description of the type of information it represents. Hence character's *title*, *strength* and *charisma* are names of data items; "*Fred the Invincible*", *3*, and *9* are examples of the actual values which might be given to these data items.

In programming, a data item is often referred to as a **variable**. This term arises from the fact that, although the name assigned to a data item cannot change, its value may vary. For example, the value assigned to a variable called *lives remaining*, will be reduced if the player's character is killed.

Activity 1.24

List the names of four data items that might be held about a player in a game of Monopoly.

Operations on Data

There are four basic operations that a computer can do with data. These are:

Input

This involves being given a value for a data item. For example, in our number-guessing game, the player who has thought of the original number is given the value of the guess from the second player. When playing Noughts and Crosses adding an X (or O) changes the set up on the board. When using a computer, any value entered at the keyboard, or any movement or action dictated by a mouse or joystick would be considered as data entry. This type of action is known as an **input operation**.

Calculation

Most games involve some basic arithmetic. In Monopoly, the banker has to work out how much change to give a player buying a property. If a character in an adventure game is hit, points must be deducted from his strength value. This type of instruction is referred to as a **calculation operation**.

Comparison

Often values have to be compared. For example, we need to compare the two numbers in our guessing game to find out if they are the same. This is known as a **comparison operation**.

Output

The final requirement is to communicate with others to give the result of some calculation or comparison. For example, in the guessing game player 1 communicates with player 2 by saying either that the guess is *Correct*, *Too high* or *Too low*.

In a computer environment, the equivalent operation would normally involve displaying information on a screen or printing it on paper. For instance, in a racing game your speed and time will be displayed on the screen. This is called an **output operation**.

Activity 1.25

Identify input, calculation, comparison and output operations when playing Hangman
For example, the algorithm needs to compare the letter guessed by the player with the letters in the word.

When describing a calculation, it is common to use arithmetic operator symbols rather than English. Hence, instead of writing the word *subtract* we use the minus sign (-). A summary of the operators available are given in TABLE-1.5.

TABLE-1.5

Mathematical Operators

English	Symbol
Multiply	*
Divide	/
Add	+
Subtract	-

Similarly, when we need to compare values, rather than use terms such as is less than, we use the less than symbol (<). A summary of these relational operators is given in TABLE-1.6.

TABLE-1.6

Relational Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

As well as replacing the words used for arithmetic calculations and comparisons with symbols, the term *calculate* or *set* is often replaced by the shorter but more cryptic symbol := between the variable being assigned a value and the value itself. Using this abbreviated form, the instruction:

```
Calculate time to complete course as distance divided by speed
```

becomes

```
time := distance / speed
```

Although the long-winded English form is more readable, this more cryptic style is briefer and is much closer to the code used when programming a computer.

Below we compare the two methods of describing our guessing game; first in English:

```
Player 1 thinks of a number between 1 and 100
REPEAT
      Player 2 makes an attempt at guessing the number
      IF guess matches number THEN
            Player 1 says "Correct
      ELSE
            IF guess is less than number THEN
                  Player 1 says "Too low"
            ELSE
                  Player 1 says "Too high"
            ENDIF
      ENDIF
UNTIL player 2 guesses correctly
```

Using some of the symbols described earlier, we can rewrite this as:

```
Player 1 thinks of a number between 1 and 100
REPEAT
      Player 2 makes an attempt at guessing the number
      IF guess = number THEN
            Player 1 says "Correct
      ELSE
            IF guess < number THEN
                  Player 1 says "Too low"
            ELSE
                  Player 1 says "Too high"
            ENDIF
      ENDIF
UNTIL guess = number
```

Activity 1.26

1. What are the two main characteristics of any data item?

2. When data is input, from where is its value obtained?

3. Give an example of a relational operator.

Levels of Detail

When we start to write an algorithm in English, one of the things we need to consider is exactly how much detail should be included. For example, we might describe how to record a programme on a video recorder as:

```
Put new tape in video
Set timer details
```

However, this lacks enough detail for anyone unfamiliar with the operation of the machine. We could replace the first statement with:

```
Press the eject button
IF there is a tape in the machine THEN
      Remove it
ENDIF
Place the new tape in the machine
```

and the second statement could be substituted by:

```
Switch to timer mode
Enter start time
Enter finish time
Select channel
```

This approach of starting with a less detailed sequence of instructions and then, where necessary, replacing each of these with more detailed instructions can be used to good effect when tackling long and complex problems.

By using this technique, we are defining the original problem as an equivalent sequence of simpler tasks before going on to create a set of instructions to handle each of these simpler problems. This divide-and-conquer strategy is known as **stepwise refinement**. The following is a fully worked example of this technique:

Problem:

Describe how to make a cup of tea.

Outline Solution:

1. Fill kettle
2. Boil water
3. Put tea bag in teapot
4. Add boiling water to teapot
5. Wait 1 minute
6. Pour tea into cup
7. Add milk and sugar to taste

This is termed a **LEVEL 1 solution**.

As a guideline we should aim for a LEVEL 1 solution with between 5 and 12 instructions. Notice that each instruction has been numbered. This is merely to help with identification during the stepwise refinement process.

Before going any further, we must assure ourselves that this is a correct and full (though not detailed) description of all the steps required to tackle the original problem. If we are not happy with the solution, then changes must be made before going any further.

Next, we examine each statement in turn and determine if it should be described in more detail. Where this is necessary, rewrite the statement to be dealt with, and below it, give the more detailed version. For example. *Fill kettle* would be expanded thus:

1. Fill kettle
    1.1 Remove kettle lid
    1.2 Put kettle under tap
    1.3 Turn on tap
    1.4 When kettle is full, turn off tap
    1.5 Place lid back on kettle

The numbering of the new statement reflects that they are the detailed instructions pertaining to statement 1. Also note that the number system is not a decimal fraction so if there were to be many more statements they would be numbered 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, etc.

It is important that these sets of more detailed instructions describe how to perform only the original task being examined - they must achieve no more and no less. Sometimes the detailed instructions will contain control structures such as IFs, WHILEs or FORs. Where this is the case, the whole structure must be included in the detailed instructions for that task.

Having satisfied ourselves that the breakdown is correct, we proceed to the next statement from the original solution.

2. Boil water
    2.1 Plug in kettle
    2.2 Switch on power at socket
    2.3 Switch on power at kettle
    2.4 When water boils switch off kettle

The next two statements expand as follows:

3. Put tea bag in teapot
    3.1 Remove lid from teapot
    3.2 Add tea bag to teapot

4. Add boiling water to teapot
    4.1 Take kettle over to teapot
    4.2 Add required quantity of water from kettle to teapot

But not every statement from a level 1 solution needs to be expanded. In our case there is no more detail to add to the statement

```
5. Wait 1 minute
```

and therefore, we leave it unchanged.

The last two statements expand as follows:

```
6. Pour tea into cup
      6.1 Take teapot over to cup
      6.2 Pour required quantity of tea from teapot into cup

7. Add milk and sugar as required
      7.1 IF milk is required THEN
      7.2         Add milk
      7.3 ENDIF
      7.4 IF sugar is required THEN
      7.5         Add sugar
      7.6         Stir tea
      7.7 ENDIF
```

Notice that this last expansion (step 7) has introduced IF statements. Control structures (i.e. IF, WHILE, FOR, etc.) can be introduced at any point in an algorithm.

Finally, we can describe the solution to the original problem in more detail by substituting the statements in our LEVEL 1 solution by their more detailed equivalent:

```
1.1 Remove kettle lid
1.2 Put kettle under tap
1.3 Turn on tap
1.4 When kettle is full, turn off tap
1.5 Place lid back on kettle
2.1 Plug in kettle
2.2 Switch on power at socket
2.3 Switch on power at kettle
2.4 When water boils switch off kettle
3.1 Remove lid from teapot
3.2 Add tea bag to teapot
4.1 Take kettle over to teapot
4.2 Add required quantity of water from kettle to teapot
5.   Wait 1 minute
6.1 Take teapot over to cup
6.2 Pour required quantity of tea from teapot into cup
7.1 IF milk is required THEN
7.2         Add milk
7.3 ENDIF
7.4 IF sugar is required THEN
7.5         Add sugar
7.6         Stir tea
7.7 ENDIF
```

This is a LEVEL 2 solution. Note that a level 2 solution includes any LEVEL 1 statements which were not given more detail (in this case, the statement `Wait 1 minute`).

For some more complex problems it may be necessary to repeat this process to more levels before sufficient detail is achieved. That is, statements in LEVEL 2 may need to be given more detail in a LEVEL 3 breakdown.

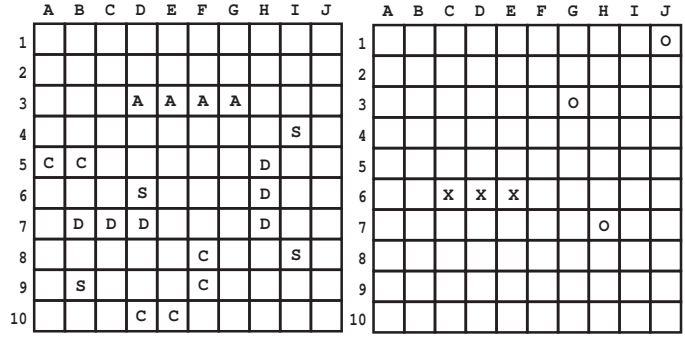
Activity 1.27

The game of battleships involves two players. Each player draws two 10 by 10 grids. Each of these have columns lettered A to J and rows numbered 1 to 10. In the first grid each player marks squares in the first grid to mark the position of warships. Ships are added as follows

1 aircraft carrier	4 squares
2 destroyers	3 squares each
3 cruisers	2 squares each
4 submarines	1 square each

The squares of each ship must be adjacent and must be vertical or horizontal.

The first player now calls out a grid reference. The second player responds to the call by saying HIT or MISS. HIT is called if the grid reference corresponds to a position of a ship. The first player then marks this result on his second grid using an o to signify a miss and x for a hit (see diagram below).



Vessels are positioned in the left-hand grid

Results of guesses are placed in the right-hand grid

If the first player achieves a HIT then he continues to call grid references until MISS is called. In response to a HIT or MISS call the first player marks the second grid at the reference called: 0 for a MISS, X for a HIT.

When the second player responds with MISS the first player's turn is over, and the second player has his turn.

The first player to eliminate all segments of the opponent's ships is the winner. However, each player must have an equal number of turns, and if both sets of ships are eliminated in the same round the game is a draw.

The algorithm describing the task of one player is given in the instructions below. Create a LEVEL 1 algorithm by assembling the lines in the correct order, adding line numbers to the finished description.

Add ships to left grid
Call grid position(s)
REPEAT
Respond to other player's call(s)
Draw grids
UNTIL there is a winner

continued on next page

Activity 1.27 (continued)

To create a LEVEL 2 algorithm, some of the above lines will have to be expanded to give more detail. More detailed instructions are given below for the statements *Call grid position(s)* and *Respond to other player's call(s)*. By reordering and numbering the lines below create LEVEL 2 details for these two statements

UNTIL other player misses
Mark position in second grid with X
Get other player's call
Get reply
Get reply
ENDIF
Call HIT
Call MISS
Mark position in second grid with 0
WHILE reply is HIT DO
Call grid reference
Call grid reference
IF other player's call matches position of ship THEN
ENDWHILE
REPEAT
ELSE

Checking for Errors

Once we've created our algorithm we would like to make sure it is correct. Unfortunately, there is no foolproof way to do this! But we can at least try to find any errors or omissions in the set of instructions we have created.

We do this by going back to the original description of the task our algorithm is attempting to solve. For example, let's assume we want to check our number guessing game algorithm. In the last version of the game we allowed the second player to make as many guesses as required until he came up with the correct answer. The first player responded to each guess by saying either "too low", "too high" or "correct".

To check our algorithm for errors we must come up with typical values that might be used when carrying out the set of instructions and those values should be chosen so that each possible result is achieved at least once.

So, as well as making up values, we need to predict what response our algorithm should give to each value used. Hence, if the first player thinks of the value 42 and the second player guesses 75, then the first player will respond to the guess by saying "Too high".

Our set of test values must evoke each of the possible results from our algorithm. One possible set of values and the responses are shown in TABLE-1.7.

TABLE-1.7

Test Data for the Number Guessing Game Algorithm

Test Data	Expected Results
number = 42	
guess = 75	Says "Too high"
guess = 15	Says "Too low"
guess = 42	Says "Correct"

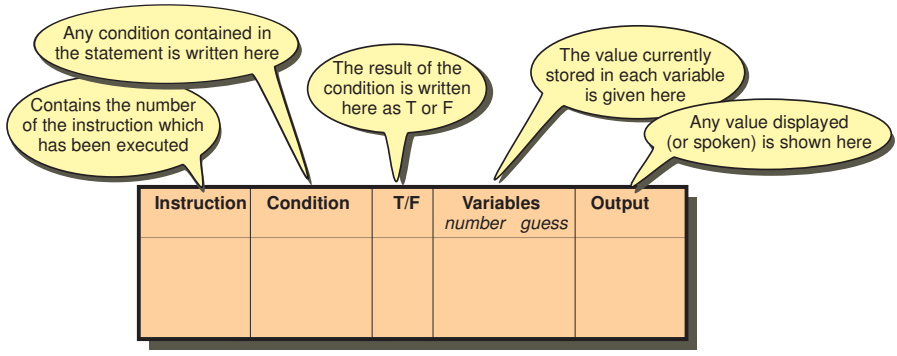
Once we've created test data, we need to work our way through the algorithm using that test data and checking that we get the expected results. The algorithm for the number game is shown below, this time with instruction numbers added.

```
1. Player 1 thinks of a number between 1 and 100
2. REPEAT
3.     Player 2 makes an attempt at guessing the number
4.     IF guess = number THEN
5.       Player 1 says "Correct"
6.     ELSE
7.       IF guess < number THEN
8.         Player 1 says "Too low"
9.       ELSE
10.        Player 1 says "Too high"
11.      ENDIF
12.    ENDIF
13. UNTIL guess = number
```

Next we create a new table (called a **trace table**) with the headings as shown in FIG-1.14.

FIG-1.14

The Components of a Trace Table



Now we work our way through the statements in the algorithm filling in a line of the trace table for each instruction.

Instruction 1 is for player 1 to think of a number. Using our test data, that number will be 42, so our trace table starts with the line shown in FIG-1.15.

FIG-1.15

Tracing the First Statement

Instruction	Condition	T/F	Variables *number guess*	Output
1			42	

The REPEAT word comes next. Although this does not cause any changes, nevertheless a 2 should be entered in the next line of our trace table. Instruction 3 involves player 2 making a guess at the number (this guess will be 75 according to our test data). After 3 instructions our trace table is as shown in FIG-1.16.

FIG-1.16

Moving through the Trace

Instruction	Condition	T/F	Variables *number guess*	Output
1			42	
2				
3			75	

Instruction 4 is an IF statement containing a condition. This condition and its result are written into columns 2 and 3 as shown in FIG-1.17.

FIG-1.17

Tracing a Condition

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			

Because the condition is false, we now jump to instruction 6 (the ELSE line) and on to 7. This is another IF statement and our table now becomes that shown in FIG-1.18.

FIG-1.18

Tracing a Second Condition

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			

Since this second IF statement is also false, we move on to statements 9 and 10. Instruction 10 causes output (speech) and hence we enter this in the final column as shown in FIG-1.19.

FIG-1.19

Recording Output

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high

Now we move on to statements 11,12 and 13 as shown in FIG-1.20.

FIG-1.20

Reaching the end of the REPEAT .. UNTIL Structure

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			

Since statement 13 contains a condition which is false, we return to statement 2 and then onto 3 where we enter 15 as our second guess (see FIG-1.21).

FIG-1.21

Showing Iteration

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			
2					
3				15	

This method of checking is known as **desk checking** or **dry running**.

Activity 1.28

Create your own trace table for the number-guessing game and, using the same test data as given in TABLE-1.7 complete the testing of the algorithm.

Were the expected results obtained?

Summary

- Computers can perform many tasks by executing different programs.
- An algorithm is a sequence of instructions which solves a specific problem.
- A program is a sequence of computer instructions which usually manipulates data and produces results.
- Three control structures are used in programs :
  - Sequence
  - Selection
  - Iteration
- A sequence is a list of instructions which are performed one after the other.
- Selection involves choosing between two or more alternative actions.
- Selection is performed using the IF statement.
- There are three forms of IF statement:

```
IF condition THEN
      instructions
ENDIF

IF condition THEN
      instructions
ELSE
      instructions
ENDIF
```

```
IF
    condition 1:
        instructions
    condition 2:
        instructions
    condition x :
        instructions
    ELSE
        instructions
ENDIF
```

- Iteration is the repeated execution of one or more statements.
- Iteration is performed using one of three instructions:

```
FOR number of iterations required DO
    instructions
ENDFOR

REPEAT
    instructions
UNTIL condition

WHILE condition DO
    instructions
ENDWHILE
```

- A condition is an expression which is either true or false.
- Simple conditions can be linked using AND or OR to produce a complex condition.
- The meaning of a condition can be reversed by adding the word NOT.
- Data items (or variables) hold the information used by the algorithm.
- Data item values may be:

```
    Input
    Calculated
    Compared
or  Output
```

- Calculations can be performed using the following arithmetic operators:

Multiplication	*	Addition	+
Division	/	Subtraction	-

- The order of priority of an operator may be overridden using parentheses.
- Comparisons can be performed using the relational operators:

Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	<>

- The symbol := is used to assign a value to a data item. Read this symbol as *is*

assigned the value.

- In programming, a data item is referred to as a variable.
- The divide-and-conquer strategy of stepwise refinement can be used when creating an algorithm.
- LEVEL 1 solution gives an overview of the sub-tasks involved in carrying out the required operation.
- LEVEL 2 gives a more detailed solution by taking each sub-task from LEVEL 1 and, where necessary, giving a more detailed list of instructions required to perform that sub-task.
- Not every statement needs to be broken down into more detail.
- Further levels of detail may be necessary when using stepwise refinement for complex problems.
- Further refinement may not be required for every statement.
- An algorithm can be checked for errors or omissions using a trace table.

Solutions

Activity 1.1

No solution required.

Activity 1.2

One possible solution is:

```
Fill A
Fill B from A
Empty B
Empty A into B
Fill A
Fill B from A
```

Activity 1.3

1. An algorithm
2. A Computer program
3. mips (millions of instructions per second)

Activity 1.4

```
Choose club
Take up correct stance beside ball
Grip club correctly
Swing club backwards
Swing club forwards, attempting to hit ball
```

The second and third statements could be interchanged.

Activity 1.5

```
Player 1 thinks of a number
Player 2 makes a guess at the number
IF guess matches number THEN
  Player 1 says "Correct"
ENDIF
Player 1 states the value of the number
```

Activity 1.6

```
IF letter appears in word THEN
  Add letter at appropriate position(s)
ELSE
  Add part to hanged man
ENDIF
```

Activity 1.7

```
IF the crossbow is on target THEN
  Say "Fire"
ELSE
  IF the crossbow is pointing too high THEN
      Say "Down a bit"
  ELSE
      IF the crossbow is pointing too low THEN
          Say "Up a bit"
      ELSE
          IF crossbow is too far left THEN
              Say "Right a bit"
          ELSE
              Say "Left a bit"
          ENDIF
      ENDIF
  ENDIF
ENDIF
```

Activity 1.8

```
IF
  you know the phrase:
      Make guess at phrase
  there are many unseen letters:
      Guess a consonant
  ELSE
      Buy a vowel
ENDIF
```

Activity 1.9

Other possibilities are:

Both conditions are true
condition 1 is true and condition 2 is false

Activity 1.10

```
IF you are first to place your hand over
those cards AND the last two cards laid
down are of the same value
THEN
  You win the cards already played
ENDIF
```

Activity 1.11

```
IF  double thrown OR fine paid THEN
  Player gets out of jail
ENDIF
```

Activity 1.12

Assuming the player has one Ace and one Knave the statement

```
IF a player has an Ace AND player has
King OR player has two Knaves
THEN
```

would reduce to

```
  IF true AND false OR false THEN
```

The AND operation is then performed giving:

```
  IF  false OR false THEN
```

Next, the OR operation is completed giving a final value of

```
  IF false THEN
```

and, therefore the player does not pick up an extra card.

Activity 1.13

```
IF (total of cards held is 43 OR hand has
4 cards of the same value ) AND hand
contains a Queen THEN
```

Activity 1.14

1. Sequence
   Selection
   Iteration

2. Boolean expression

3. Binary selection
   Multi-way selection

4. No more than one of the conditions can be true at any given time.

5. Boolean operators are: AND, OR, and NOT.

6. AND is performed before OR .

7. The order in which operations in a complex condition are calculated can changed by using parentheses.

Activity 1.15

```
Throw dice
Add dice value to total
```

Activity 1.16

Only one line, the FOR statement, would need to be changed, the new version being:

```
  FOR 10 times DO
```

To call out the average, the algorithm would change to

```
Set the total to zero
FOR 10 times DO
  Throw dice
  Add dice value to total
ENDFOR
Calculate average as total divided by 10
Call out the value of average
```

Activity 1.17

In fact, only the first line of our algorithm is not repeated, so the lines that need to be repeated are:

```
Player 2 makes an attempt at guessing the
number
IF guess matches number THEN
  Player 1 says "Correct "
ELSE
  IF guess is less than number THEN
     Player 1 says "Too low"
  ELSE
     Player 1 says "Too high"
  ENDIF
ENDIF
```

Activity 1.18

The FOR loop forces the loop body to be executed exactly 7 times. If the player guesses the number in less attempts, the algorithm will nevertheless continue to ask for the remainder of the 7 guesses.

Later, we'll see how to solve this problem.

Activity 1.19

```
FOR 6 times DO
  Pick out ball
  Call out number on the ball
ENDFOR
```

Activity 1.20

```
FOR every card in player's hand DO
  IF card is a knight THEN
     Remove card from hand
  ENDIF
ENDFOR
```

Activity 1.21

```
REPEAT
  Place coin in machine
  Pull arm
  IF a win THEN
     Collect winnings
  ENDIF
UNTIL all coins are gone OR winnings are
at least £10.00
```

Activity 1.22

```
Roll both dice
WHILE both dice do not match in value DO
  Choose dice with lower value
  Roll the chosen dice
ENDWHILE
```

Activity 1.23

1. Iteration means executing a set of instructions over and over again.
2. The three looping structures are:

```
FOR .. ENDFOR
REPEAT .. UNTIL
WHILE .. ENDWHILE
```

3. The FOR .. ENDFOR structure.
4. The WHILE .. ENDWHILE structure.
5. The REPEAT .. UNTIL structure.

Activity 1.24

Number of properties held
Amount of money held
The playing token being used
The position on the board

Activity 1.25

Input:
- Letter guessed
- Word guessed

Calculations:
- Where to place a correctly guessed letter
- The number of wrong guesses made

Comparisons:
- The letter guessed with the letters in the word
- The word guessed with the word to be guessed
- The number of wrong guesses with the value 6 (6 wrong guesses completes the drawing of the hanged man)

Output:
- Hyphens indicating each letter in the word
- Gallows
- Body parts of the hanged man
- Correctly guessed letters

Activity 1.26

1. Name and value
2. From outside the system. In a computerised system this is often via a keyboard.
3. The relational operators are:
   <, <=, >, >=, =, and <>

Activity 1.27

The LEVEL 1 is coded as:

```
1. Draw grids
2. Add ships to left grid
3. REPEAT
4.      Call grid position(s)
5.      Respond to other player's call(s)
6. UNTIL there is a winner
```

The expansion of statement 4 would become:

```
4.1 Call grid reference
4.2 Get reply
4.3 WHILE reply is HIT DO
4.4     Mark position in second grid with X
4.5     Call grid reference
4.6      Get reply
4.7 ENDWHILE
4.8 Mark position in second grid with 0
```

The expansion of statement 5 would become:

```
5.1.REPEAT
5.2     Get other player's call
5.3     IF other player's call matches position of ship THEN
5.4         Call HIT
5.5     ELSE
5.6         Call MISS
5.7     ENDIF
5.8 UNTIL other player misses
```

Activity 1.28

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			
2					
3				15	
4	guess = number	F			
6					
7	guess < number	T			
8					Too low
11					
12					
13	guess = number	F			
2					
3				42	
4	guess = number	T			
5					Correct
11					
12					
13	guess = number	T			

The expected results were obtained.

2

Starting DarkBASIC Pro

Correcting Errors

Creating a Project in DarkBASIC Pro

Executing a Program

Screen Output

Text Colour, Size, Font, and Style

The Compilation Process

Transparent and Opaque Text

Using the DarkBASIC Pro Editor

Programming a Computer

Introduction

In the last chapter we created algorithms written in a style of English known as structured English. But if we want to create an algorithm that can be followed by a computer, then we need to convert our structured English instructions into a programming language.

There are many programming languages; C++, Java, C#, and Visual Basic being amongst the most widely used. So how do we choose which programming language to use? Probably the most important consideration is the area of programming that is best suited to a given language. For example, Java is designed to create programs that can be executed on a variety of different computers, while C++ was designed for fast execution times.

We are going to use a language known as DarkBASIC Professional or just DarkBASIC Pro, which was designed specifically for writing computer games. Because of this, it has many unique commands for displaying graphics, controlling joysticks, and creating three dimensional images.

The Compilation Process

As we will soon see, DarkBASIC Pro uses statements that retain some English terms and phrases, so we can look at the set of instructions and make some sense of what is happening after only a relatively small amount of training.

Binary is a method of representing numbers using only the digits 0 and 1.

Unfortunately, the computer itself only understands instructions given in a **binary code** known as **machine code** and has no capability of directly following a set of instructions written in DarkBASIC Pro. But this need not be a problem. If we were given a set of instructions written in Russian we could easily have them translated into English and then carry out the translated commands.

This is exactly the approach the computer uses. We begin the process of creating a new piece of software by mentally converting our structured English into DarkBASIC Pro commands. These commands are entered using a **text editor** which is nothing more than a simple word-processor-like program allowing such basic operations as inserting and deleting text. Once the complete program has been entered, we get the machine itself to translate those instructions into machine code. The original code is known as the **source code**; the machine code equivalent is known as the **object code**.

The translator (known as a **compiler**) is simply another program installed in the computer. After typing in our program instructions, we feed these to the compiler which produces the equivalent instructions in machine code. These instructions are then executed by the computer and we should see the results of our calculations appear on the screen (assuming there are output statements in the program).

The compiler is a very exacting task master. The structure, or **syntax**, of every statement must be exactly right. If you make the slightest mistake, even something as simple as missing out a comma or misspelling a word, the translation process will fail. When this happens in DarkBASIC Pro the incorrect command is highlighted in red.

A failure of this type is known as a **syntax error** - a mistake in the grammar of your commands. Any syntax errors have to be corrected before you can try compiling the program again.

As we work on the computer entering a DarkBASIC Pro program, we need to save this source code to a file. This ensures that we have a copy of our work should there be a power cut or we accidentally delete the program from the computer's memory. DarkBASIC Pro refers to this as the **source file**.

But a second file, known as the **project file** is also produced. This second file is created automatically by DarkBASIC Pro and contains details of any images, sounds or other resources that might be used by your program.

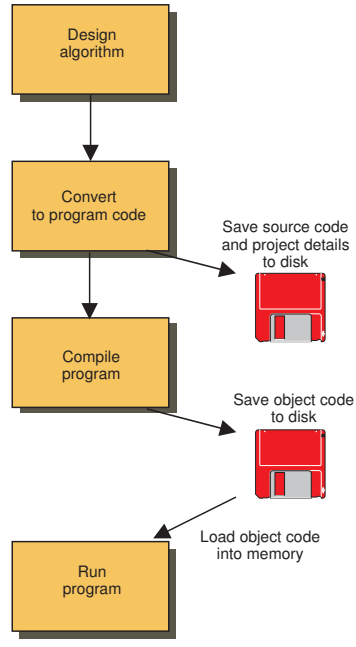
When we compile our program (translating it from source code to object code), yet another file is produced. This third file, the **executable file**, contains the object code and is, again, created automatically.

To run our program, the source code in the executable file is loaded into the computer's memory (RAM) and the instructions it contains are carried out.

The whole process is summarised in FIG-2.1.

FIG-2.1

Creating Software



If we want to make changes to the program, we load the source code into the editor, make the necessary changes, then save and recompile our program, thereby replacing the old version of all three files.

Activity 2.1

1. What type of instructions are understood by a computer?

2. What piece of software is used to translate a program from source code to object code?

3. Misspelling a word in your program is an example of what type of error?

Starting DarkBASIC Pro

Introduction

DarkBASIC Pro is based on one of the earliest computer languages, BASIC, but has been enhanced specifically to aid the creation of games programs.

The language was invented by Lee Bamber who formed a company to sell DarkBASIC Pro. Over the last few years the company has grown in size and expanded to sell other DarkBASIC related products, such as DarkMatter, which contains many 3D objects that can be used in DarkBASIC programs.

In fact, there are two versions of the language: DarkBASIC and DarkBASIC Professional. It's this second, enhanced version of the language we will be using here.

DarkBASIC Pro Files

Because a typical program written in DarkBASIC Pro is likely to contain images, sounds and even video, the DarkBASIC Pro package has to save much more than the set of instructions that make up your program; it also needs to store details of these images, sounds, etc.

To do this DarkBASIC Pro creates two files every time you produce a new program (see FIG-2.2).

The first of these files, known as the **project file**, contains details of the images and sounds used by your program, as well as other information such as the screen resolution and number of colours used. This file has a *.dbpro* extension.

The second file, known as the **source file**, contains only the program's code written in the DarkBASIC Pro language. This file has a *.dba* extension.

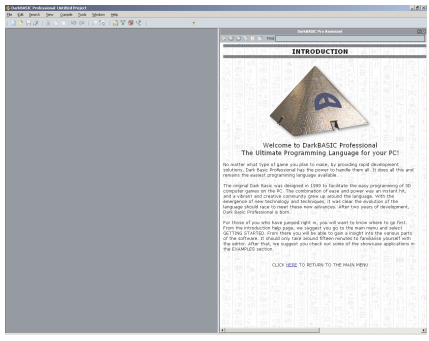
FIG-2.2

The Two Files Created by a DarkBASIC Pro Program

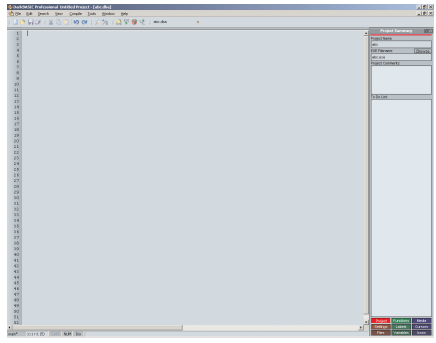
When you first start up DarkBASIC Pro you should see one of the screens shown in FIG-2.3. Exactly which one you see depends on how often DarkBASIC Pro has been run on your computer. The first time the program is run, the display will match that shown on the left of FIG-2.3; every other time your screen will match that shown on the right.

FIG-2.3

The Start-Up Screen in DarkBASIC Pro



DarkBASIC Pro Start-Up Screen (First Start-Up Only)



DarkBASIC Pro Start-Up Screen (Subsequent Start-Ups)

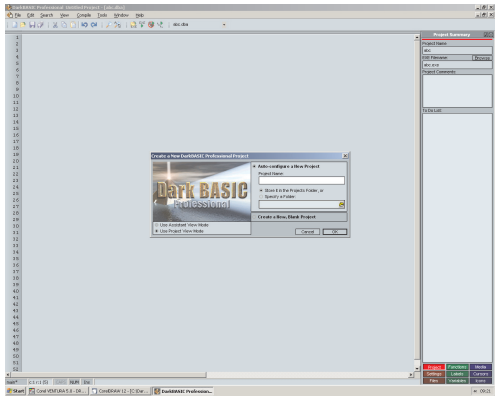
First Start-Up

If this is the first time DarkBASIC Pro has been run on your machine, as well as the main window, the **Assistant Window** also shows on the right-hand side.

If you close down the Assistant Window the display changes to match that shown in FIG-2.4, showing the Project Dialog box.

FIG-2.4

The Project Dialog Box



Subsequent Start-Ups

When DarkBASIC Pro is started up for the second (or subsequent) time, use the FILE | NEW PROJECT option from the main menu, or click on the *New Project* icon near the top left corner, to display the *Project Dialog* box.

Specifying a Project

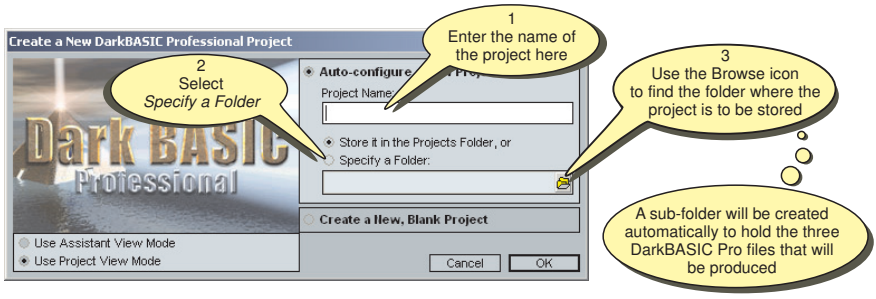
The next stage is to create a project file by filling in the details required by the Project Dialog box.

First the name to be given to the project is entered. This should be something meaningful like *Hangman* or *SpaceMonsters*.

Next the *Specify a Folder* radio button is selected and the folder in which the DarkBASIC Pro projects are to be saved is entered. The folder specified must already exist. See FIG-2.5 for a summary of these steps.

FIG-2.5

Filling in the *New Project* Dialog Box



Once the OK button in the Project Dialog box is clicked, the dialog box disappears and you are left with the main edit area where the program code is entered. Line numbers appear to the left of this area.

A First Program

Before we begin looking in detail at the commands available in DarkBASIC Pro, we'll have a quick look at a simple program and show you how to type it in, run it and save the code.

The program in LISTING-2.1 gets you to enter your name at the keyboard and then displays a greeting on the screen.

LISTING-2.1

A First Program

```
Rem Project: First
Rem Created: 02/10/2004 07:35:27
Rem ***** Main Source File *****

REM *** A program to read and display your name ***
INPUT "Enter your name : ",name$
PRINT "Hello " ,name$, " welcome to DarkBASIC Pro."
WAIT KEY
END
```

An Explanation of the Code

DarkBASIC Pro allows words to be given in either upper or lower case.

REM

This is short for REMARK and is used to indicate a comment within the program. Comments are totally ignored when the source code is translated into object code and are only included for the benefit of anybody examining the program code, giving an explanation of what the program does.

When you type in a program, you'll see that the instructions are colour-coded with keywords appearing in blue.

INPUT

This is a keyword in DarkBASIC Pro. Keywords are words recognised by the programming language as having a specific meaning.

All keywords are shown throughout this text in uppercase, but lowercase characters are also acceptable.

The INPUT keyword tells the computer to allow the user to enter a value from the keyboard.

24

Sound

Creating 3D Sound

Loading and Playing Mono and Stereo WAV Files

Positioning a 3D sound

Positioning the Listener

Rotating the Listener

Setting Sound Speed

Setting Sound Volume

Mono and Stereo Sound

Introduction

Don't confuse sound files with music files. The set of instructions for handling sound files is much more extensive than that for music (which we looked at in Chapter 15). The position of a sound can be moved around in 3-dimensional space, giving the listener a total surround-sound experience. Even the apparent position of the listener can be moved.

DarkBASIC Pro sound commands only handle WAV files. To play other file types use either the MUSIC or ANIMATION commands.

WAV files can be recorded in mono (1 track) or stereo (2 tracks). Stereo sound has the advantage of retaining spatial information about sound sources, but creates files twice the size of the equivalent mono file. The commands given below apply equally to each type of recording.

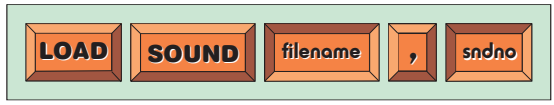
The Basics of Loading and Playing Sounds

The LOAD SOUND Statement

Before using a WAV file, it must be loaded using the LOAD SOUND statement which has the format shown in FIG-24.1.

FIG-24.1

The LOAD SOUND Statement



In the diagram:

filename — is a string specifying the name of the file to be loaded. It is best if the file has been previously copied to your directory using the **Media|Add** option. However, path information can be included in the string.

sndno — is an integer value by which the sound is identified within the program. Any positive integer value can be used, although no two sounds can use the same value at the same time.

A typical example of using this statement would be:

```
LOAD SOUND "help.wav", 1
```

If the requested file is not found or cannot be loaded for some reason, then your program will terminate.

The PLAY SOUND Statement

Once the sound file has been loaded, it can be played using the PLAY SOUND statement. In its simplest form we need only state the sound number of the file we

want to play. For example, assuming we've loaded a sound file and assigned it the number 1, we can play that sound with the statement:

```
PLAY SOUND 1
```

The PLAY SOUND statement causes the sound file to start playing, but subsequent statements in your program will continue to be executed while the sound plays.

LISTING-24.1 gives a simple example of how to play a sound file.

LISTING-24.1

Playing a Sound File

```
REM *** Load sound ***
LOAD SOUND "welcome.wav", 1
WAIT 500
REM *** Play sound ***
PLAY SOUND 1
REM *** End program ***
WAIT KEY
END
```

The WAIT 500 statement gives the program time to complete the LOAD instruction before attempting to play the file. Without this, it is possible that the first part of the sound will be missing when it is played.

Activity 24.1

Type in and test the program given in LISTING-24.1 (*sounds01.dbpro*).

Make sure you've copied the required sound file to your folder and that sound is enabled on your computer.

Sound files need not be played from the start. Instead, by using an extended version of the PLAY SOUND statement, you can specify how many bytes into the file playing should begin. So, if we don't want to play the first 10000 bytes (bytes 0 to 9999) of the sound file, we could use the statement:

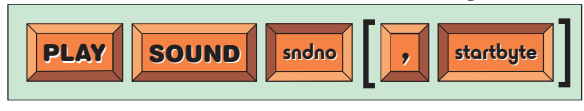
```
PLAY SOUND 1, 10000
```

You'll probably have to play about with the byte value until you get the starting point you want.

The complete format of the PLAY SOUND statement is given in FIG-24.2.

FIG-24.2

The PLAY SOUND Statement



In the diagram:

sndno	is the integer previously assigned to the sound file to be played.
startbyte	is an integer representing the number of bytes into the file at which playing is to start. If omitted, a value of zero is assumed.

Activity 24.2

Modify your last program so that it begins playing at byte 20000.

The LOOP SOUND Statement

Whereas the PLAY SOUND statement plays the sound file only once, the LOOP SOUND statement will replay the sound continuously. Again, there is more than one option when using this statement. The simplest of these is to state only the sound number. For example, the line

```
LOOP SOUND 1
```

will play sound 1 over and over again.

A second option is to specify a start byte in the file, as in the line:

```
LOOP SOUND 1,11000
```

This causes the first 11000 bytes of the file to be omitted on every play.

The next option allows us to stop the sound before the end of the file is reached. Typically, we might write

```
LOOP SOUND 1,11000,50000
```

This time, only the sound which occupies bytes 11000 to 50000 will be played during each loop.

If you want the first playing to start at a different position, you can add a final parameter, as in the line:

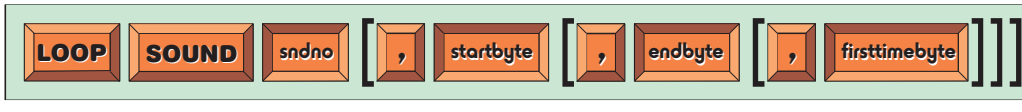
```
LOOP SOUND 1,11000,50000,30000
```

Now bytes 30000 to 50000 will play the first time round and subsequently it will be bytes 11000 to 50000 that play.

FIG-24.3

The LOOP SOUND Statement

The complete format of the LOOP SOUND statement is given in FIG-24.3.



In the diagram:

sndno	is the integer previously assigned to the sound file to be looped.
startbyte	is an integer representing the number of bytes into the file at which playing is to start on each iteration.
endbyte	is an integer representing the number of bytes into the file at which playing is to cease.
firsttimebyte	is an integer representing the number of bytes into the file at which playing is to begin on the first play only. *firsttimebyte* should have a value between *startbyte* and *endbyte*.

Activity 24.3

Modify your previous program so that the complete sound file is played continuously.

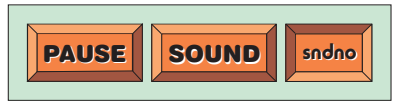
Modify the file again so that it plays bytes 20000 to 50000 only.

The PAUSE SOUND Statement

It is possible to pause a playing sound file using the PAUSE SOUND statement, which has the format shown in FIG-24.4.

FIG-24.4

The PAUSE SOUND Statement



In the diagram:

sndno	is an integer giving the number of the sound to be paused.

We could, therefore, pause the sound assigned sound number 1 with the line:

```
PAUSE SOUND 1
```

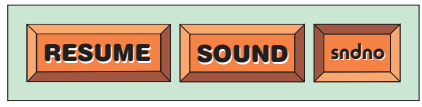
This assumes, of course, that the sound 1 is currently playing.

The RESUME SOUND Statement

A paused sound file can be resumed using the RESUME SOUND statement, which has the format shown in FIG-24.5.

FIG-24.5

The RESUME SOUND Statement



In the diagram:

sndno	is an integer giving the number of the sound to be resumed.

Of course, we should only use this statement on a sound file that has previously been paused. So, assuming we have a paused sound with the statement

```
PAUSE SOUND 1
```

then we could resume the sound using the line

```
RESUME SOUND 1
```

The sound resumes from exactly the point in the file where it paused.

The program in LISTING-24.3 pauses the playing sound when any key is pressed and resumes it when a second key is pressed. Pressing a third key terminates the program.

LISTING-24.2

Pausing and Resuming a Playing Sound

```
REM *** Load sound ***
LOAD SOUND "demo.wav", 1
WAIT 500
REM *** Play sound continuously ***
LOOP SOUND 1

REM *** Pause when key is pressed ***
WAIT KEY
PAUSE SOUND 1

REM *** Resume when key is pressed ***
WAIT KEY
RESUME SOUND 1

REM *** End program ***
WAIT KEY
END
```

Activity 24.4

Type in and test the program given above (*sounds02.dbpro*).

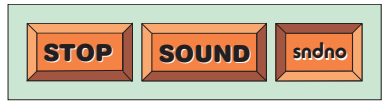
Does the sound resume looping?

The STOP SOUND Statement

Rather than just pause a sound file, we can stop it altogether using the STOP SOUND statement, which has the format shown in FIG-24.6.

FIG-24.6

The STOP SOUND Statement



In the diagram:

sndno	is an integer giving the number of the sound to be stopped.

For example, the line

```
STOP SOUND 1
```

will terminate the playing of sound 1.

However, the sound file can still be resumed using the RESUME SOUND statement. But, when resuming a stopped sound, play starts from the beginning of the file.

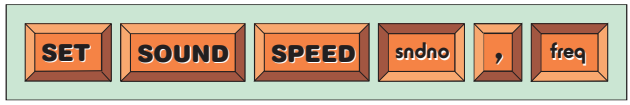
The SET SOUND SPEED Statement

By changing the speed of a sound, we can make an everyday sound resemble a clap of thunder (when the sound is slowed down) or like a high-pitched squeak (when it's speeded up).

The speed at which a sound file plays can be changed using the SET SOUND SPEED statement. The format of this statement is shown in FIG-24.7.

FIG-24.7

The SET SOUND SPEED Statement



In the diagram:

sndno	is an integer giving the number of the sound whose speed is to be set.
freq	is an integer value representing the upper frequency to be produced when the file is playing. This value must lie in the range 100 to 100,000.

A frequency value may seem a rather strange way to set the speed of a sound file, but a bit of trial and error will soon get the effect you want. Most files will sound normal with a frequency setting of 22,000, with 11,000 giving half speed and 44,000 giving double speed.

To play sound 1 at half speed we would use the lines:

```
SET SOUND SPEED 1, 11000
PLAY SOUND 1
```

Activity 24.5

Modify your previous program so that the sound file is at half speed when it is first played and double speed when it is resumed.

The SET SOUND VOLUME Statement

The SET SOUND VOLUME statement is designed to modify the volume of a sound and has the format shown in FIG-24.8.

FIG-24.8

The SET SOUND VOLUME Statement

In the diagram:

sndno	is an integer giving the number of the sound whose volume is to be set.
perc	is an integer value representing the new volume as a percentage of the original volume. This value must lie in the range 0 (silent) to 100 (normal).

Because of the percentage values allowed, it is not possible to increase the volume of the sound from its normal level, only decrease it.

For example, we could reduce the volume of a sound with the statements

```
SET SOUND VOLUME 1, 90
PLAY SOUND 1
```

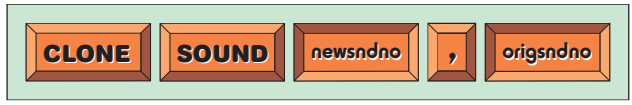
The CLONE SOUND Statement

If required, you can assign as second sound number to an already loaded sound file. This allows the second sound file to be played independently from the first, but does not require a second copy of the sound file to be held in RAM and thereby saves memory space.

A copy of an existing sound file is made using the CLONE SOUND statement which has the format shown in FIG-24.9.

FIG-24.9

The CLONE SOUND Statement



In the diagram:

newsndno	is the additional integer value to be assigned to the new sound object.
origsndno	is the integer value specifying which existing sound object is to be copied into the new sound object.

The program in LISTING-24.3 assigns two values to a sound file and then plays both sound objects with a one second delay between each.

LISTING-24.3

Cloning a Sound

```
REM *** Load sound file ***
LOAD SOUND "sample.wav", 1

REM *** Assign a second value to the sound ***
CLONE SOUND 2,1

REM *** Start playing the sound using its first value ***
PLAY SOUND 1

REM *** Wait 1 second ***
SLEEP 1000

REM *** Play the sound using its second value ***
PLAY SOUND 2

REM *** End program ***
WAIT KEY
END
```

Activity 24.6

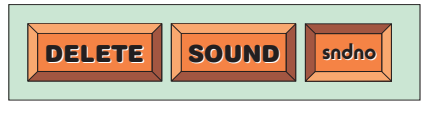
Type in and test the program given above (*sounds03.dbpro*).

The DELETE SOUND Statement

A loaded sound file occupies RAM space, so when you're finished with the sound it is best to delete it - thereby freeing up the RAM space that is holding the sound file. The DELETE SOUND statement has the format shown in FIG-24.10.

FIG-24.10

The DELETE SOUND Statement



In the diagram:

sndno	is the number of the existing sound object to be deleted.

Note that the sound object is not deleted if a clone exists. To remove the sound from RAM the original sound object and all of its clones must be deleted.

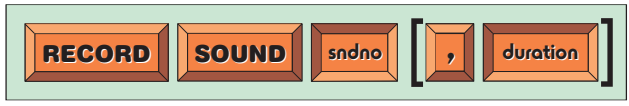
It is also possible to make a DarkBASIC Pro program record sound if you have a microphone attached to your sound card.

The RECORD SOUND Statement

To record a sound use the RECORD SOUND statement. The statement defaults to a 5 second recording, but by entering duration information, a recording of any (reasonable) length can be made. The format for the RECORD SOUND statement is given in FIG-24.11.

FIG-24.11

The RECORD SOUND Statement



In the diagram:

sndno	is the integer value to be assigned to the recording.
duration	is an integer representing duration of the recording in milliseconds.

So, the command

```
RECORD SOUND 1
```

should create sound object 1 from a 5 second recording from the microphone - but it's not quite that easy!

The RECORD SOUND statement only works if you've already used that sound number to load a file. So, the line

```
RECORD SOUND 1
```

will give us an error unless we have previously loaded a file (even though you're not going to use that file) using the sample sound number. For example:

```
LOAD "sample.wav",1
RECORD SOUND 1
```

To make a 10 second recording and assign it to sound number 3, we could use the statements

```
LOAD "mysound.wav",3
RECORD SOUND 3, 10000
```

The program will continue to execute other statements in your program while the recording is being made. Normally, you won't want this to happen, so the RECORD SOUND statement should be followed by a WAIT KEY statement or a WAIT milliseconds statement (in which the number of milliseconds matches, or is slightly greater than, the length of the recording).

The STOP RECORDING SOUND Statement

If you want to stop a recording before the specified duration has elapsed, then you

can use the STOP RECORDING SOUND statement which has the format shown in FIG-24.12.

FIG-24.12

The STOP RECORDING SOUND Statement

The program in LISTING-24.4 records from the microphone for up to 10 seconds, but stops early if a key is pressed. When the recording is complete, it is replayed.

LISTING-24.4

Recording a Sound

```
REM *** Load a sound file for sound number being used    ***
REM *** We won't use this file, but the sound number     ***
REM *** needs to be loaded first, otherwise RECORD SOUND ***
REM *** won't work                                       ***
LOAD SOUND "sample.wav",1

REM *** Start recording - 10 seconds ***
PRINT "Speak now "
RECORD SOUND 1,10000

REM *** Stop recording after 10 secs or key pressed ***
WAIT KEY
STOP RECORDING SOUND

REM *** Play back the recorded sound ***
PRINT "Replaying now..."
SLEEP 500
PLAY SOUND 1

REM *** End program ***
WAIT KEY
END
```

Activity 24.7

Plug a microphone into your system. Check that it is operating correctly (Try Accessories|Entertainment|Sound Recorder)

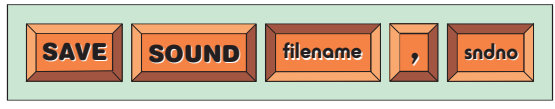
Type in the program given above (*sounds04.dbpro*) and test that it operates correctly.

The SAVE SOUND Statement

Your recorded sound can be saved to a file using the SAVE SOUND statement which has the format shown in FIG-24.13.

FIG-24.13

The SAVE SOUND Statement



In the diagram:

filename	is a string giving the file name to be used when saving the sound data. This string may include path information. The named file should not already exist.
sndno	is the integer value previously assigned to the sound that is to be saved.

We could save a recorded sound (with sound number 1) to a file named *speech.wav* in the current folder with the line:

```
SAVE SOUND "speech.wav",1
```

Activity 24.8

Modify your previous program so that the recorded sound is saved to *speech.wav* after it has been replayed.

What happens if you try to run this program a second time?

Retrieving Sound File Data

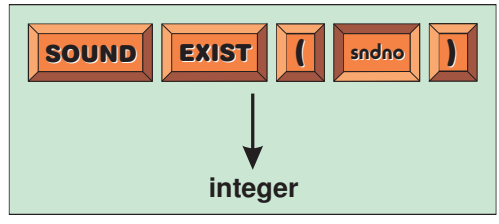
Several statements exist in DarkBASIC Pro which allow information about currently loaded sound files to be retrieved. For example, we can find out whether a sound object is playing, stopped, paused, etc. The statements that supply us with this information are described below.

The SOUND EXIST Statement

We can find out if a specific number has been assigned to a sound file by using the SOUND EXIST statement. This statement returns the value 1 if the specified number has been assigned to a sound file, otherwise zero is returned. The statement has the format shown in FIG-24.14.

FIG-24.14

The SOUND EXIST Statement



In the diagram:

sndno is the number of the sound object being checked.

Typical examples of how the statement might be used are:

```
loaded = SOUND EXIST(1)
```

and

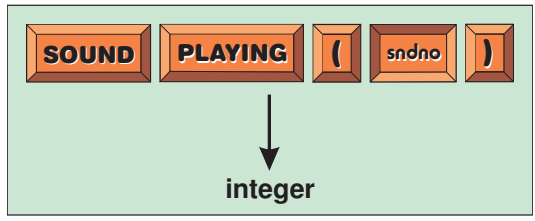
```
IF SOUND EXIST(3) = 1
    PRINT "SOUND file loaded in 3"
ELSE
    PRINT "SOUND file not loaded in 3"
ENDIF
```

The SOUND PLAYING Statement

This statement returns 1 if the sound file is currently playing, otherwise zero is returned. The statement has the format shown in FIG-24.15.

FIG-24.15

The SOUND PLAYING Statement



In the diagram:

sndno is the number of the sound object being tested.

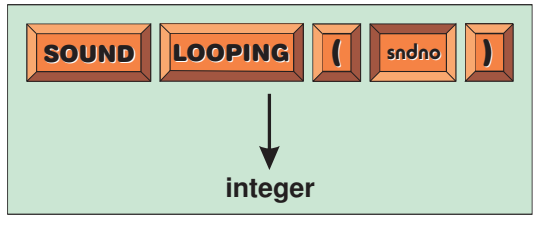
The value 1 is returned if the sound file is currently playing because of either a PLAY SOUND or LOOP SOUND statement.

The SOUND LOOPING Statement

The SOUND LOOPING statement returns 1 if the sound file is currently looping, otherwise zero is returned. The statement has the format shown in FIG-24.16.

FIG-24.16

The SOUND LOOPING Statement



In the diagram:

sndno is the number of the sound object being tested.

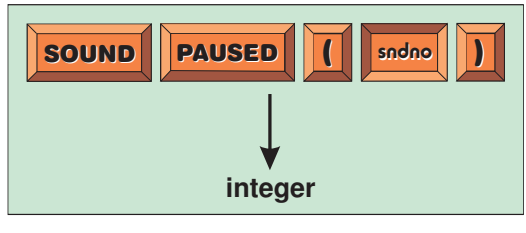
A value of 1 is only returned if the sound file is playing because of a LOOP SOUND statement. A playing sound file initiated using the PLAY SOUND statement will return zero.

The SOUND PAUSED Statement

The SOUND PAUSED statement returns 1 if the sound file is currently paused, otherwise zero is returned. The statement has the format shown in FIG-24.17.

FIG-24.17

The SOUND PAUSED Statement



In the diagram:

sndno is the number of the sound object being tested.

The program in LISTING-24.5 displays the state of the playing sound file.

LISTING-24.5

Displaying the State of a Sound File

```
REM *** Load sound file ***
LOAD SOUND "demo2.wav", 1
WAIT 500

REM *** Play sound file continuously ***
LOOP SOUND 1

REM *** Get sound file's status ***
playing = SOUND PLAYING(1)
looping = SOUND LOOPING(1)
paused = SOUND PAUSED(1)

REM *** Display details ***
IF playing = 1
   PRINT "SOUND is playing"
ENDIF
IF looping = 1
   PRINT "SOUND is looping"
ENDIF
IF paused = 1
   PRINT "SOUND is paused"
ENDIF

REM *** End program ***
WAIT KEY
END
```

Activity 24.9

Type in and test the program given above (*sounds05.dbpro*).

1) Change the line

```
LOOP SOUND 1
```

to

```
PLAY SOUND 1
```

How does this affect the messages displayed?

2) Add the lines

```
WAIT KEY
PAUSE SOUND 1
```

after the `PLAY SOUND 1` statement.

How do the messages change?

3) Change the line

```
PAUSE SOUND 1
```

to

```
STOP SOUND 1
```

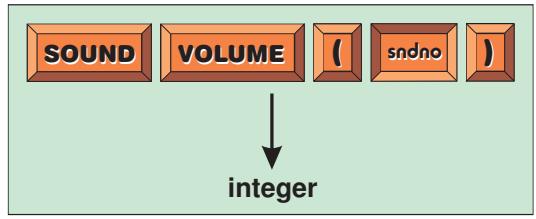
Are the messages changed?

The SOUND VOLUME Statement

This statement returns the volume setting for the specified sound. The statement's format is shown in FIG-24.18

FIG-24.18

The SOUND VOLUME Statement



In the diagram:

sndno	is the number of the sound object being interrogated.

A typical statement to determine the current volume setting for sound 1 would be:

```
volume = SOUND VOLUME(1)
```

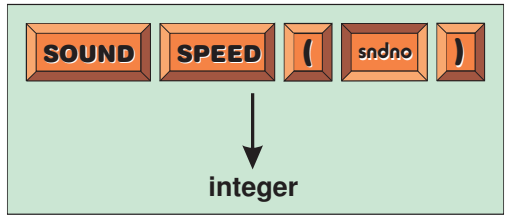
A sound file playing at default volume would return the value 100.

The SOUND SPEED Statement

This statement returns the speed setting for the specified sound. The statement's format is shown in FIG-24.19.

FIG-24.19

The SOUND VOLUME Statement



In the diagram:

sndno	is the number of the sound object being interrogated.

A typical statement to determine the current speed setting for sound 1 would be:

```
speed = SOUND SPEED(1)
```

A sound file playing at default speed would return the value 100.

Activity 24.10

Modify your previous program to display the speed and volume settings of the playing sound file.

Activity 24.11

Modify the final version of the asteroids project you created in the last chapter to play a sound when a missile is fired and an asteroid is hit.

Moving a Sound

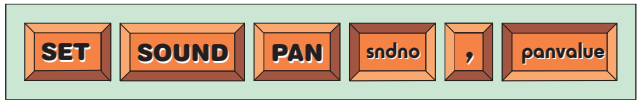
When stereo sound was first introduced, a favourite demonstration of its effects was to have a train rush past the listener. You could here it move from the left, through the centre and off to the right. In its day, it was very impressive!

The SET SOUND PAN Statement

We can create a similar effect, making a sound appear as if its coming from anywhere between our two speakers using the SET SOUND PAN statement. The statement has the format shown in FIG-24.20.

FIG-24.20

The SET SOUND PAN Statement



In the diagram:

sndno	is the number of the sound object to be panned.
panvalue	is an integer value between -10,000 and + 10,000. A value of -10,000 will make the sound come exclusively from the left speaker, zero places the sound at the centre position between the speakers, and 10,000 places the sound by the right speaker.

In LISTING-24.6 a sound is panned while it plays.

LISTING-24.6

Panning a Sound File

```
REM *** Load sound ***
LOAD SOUND "welcome.wav", 1

REM *** Play sound ***
WAIT 500
PLAY SOUND 1
REM *** Change pan setting while sound playing ***
FOR c = -10000 TO 10000 STEP 1000
   SET SOUND PAN 1,c
   WAIT 100
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 24.12

Type in and test the program in LISTING-24.7 (*sounds06.dbpro*).

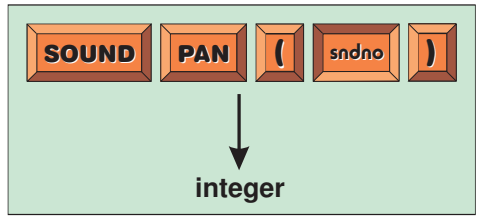
Reverse the FOR loop (making it go from 10,000 to -10,000). How does this affect the sound output?

The SOUND PAN Statement

We can determine the current pan setting using the SOUND PAN statement. This has the format shown in FIG-24.21.

FIG-24.21

The SOUND PAN Statement



In the diagram:

sndno is the number of the sound object to be tested.

The value returned will be an integer in the range -10,000 to +10,000 and will match the value assigned by the last SET SOUND PAN statement for this object. If a SET SOUND PAN has not been performed on the object, then a value of zero will be returned.

Playing Multiple Sound Files

It's possible to play two or more sound files at the same time by loading multiple sound files. For example, we could load two sound files with the lines:

```
LOAD SOUND "demo1.wav",1
LOAD SOUND "demo2.wav",2
```

These can then be played at the same time using the lines

```
PLAY SOUND 1
PLAY SOUND 2
```

The program in LISTING-24.7 demonstrates two sound files being played. The first file starts playing right away, but the second file starts playing after a key is pressed.

LISTING-24.7

Playing Multiple Sound Files

```
REM *** Load sound files ***
LOAD SOUND "demo1.wav",1
LOAD SOUND "demo2.wav",2

REM *** Play first sound file continuously ***
LOOP SOUND 1

REM *** Wait for key press, then start second file playing ***
WAIT KEY
LOOP SOUND 2

REM *** End program ***
WAIT KEY
END
```

Activity 24.13

Type in and test the program given above (*sounds07.dbpro*).

Summary

- The SOUND statements are designed to *.wav* files only.
- Use LOAD SOUND to load a sound file into RAM.

- Use PLAY SOUND to play a sound file once only.
- Use LOOP SOUND to play a sound file repeatedly.
- Use PAUSE SOUND to pause a sound file which is currently playing.
- Use RESUME SOUND to resume a sound file which has been paused.
- Use STOP SOUND to halt a playing sound file.
- Use SET SOUND SPEED to set the highest frequency (and hence the speed) of a sound file.
- Use SET SOUND VOLUME to set the volume of a sound file when it is played.
- Use CLONE SOUND to make an independent copy of a loaded sound file.
- Use DELETE SOUND to remove a sound from RAM.
- Use RECORD SOUND to initiate a sound recording.
- Use STOP RECORDING SOUND to stop the current recording.
- Use SAVE SOUND to save a sound file currently held in RAM to backing store.
- SOUND EXIST returns 1 if the specified sound file is currently in RAM.
- SOUND PLAYING returns 1 if the specified sound file is currently playing or looping.
- SOUND LOOPING returns 1 if the specified sound file is currently looping.
- SOUND PAUSED returns 1 if the specified sound file is currently paused.
- SOUND VOLUME returns the volume setting of a specified sound file.
- SOUND SPEED returns the current maximum frequency of a specified sound file.
- Use SET SOUND PAN to set the balance between the two front speakers.
- SOUND PAN returns the current balance setting of a specified sound file.
- Several sound files can be played simultaneously.

3D Sound Effects

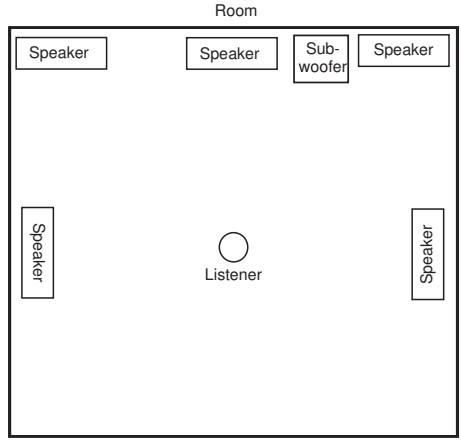
Introduction

DarkBASIC Pro contains an additional set of commands specifically for handling 3D sound effects. A 3D sound can appear to the listener to originate from anywhere in space - to the side, front, back, above, or below the listener.

The 3D effect is not recorded within the sound file, but is created by manipulating the balance, delay and loudness of the speakers connected to your sound card. A good sound card will allow you to use a 5.1 speaker system (with five speakers and a sub-woofer). The speakers should be positioned as shown in FIG-24.22. The position of the sub-woofer, which generates the very low frequency noises, is not important since our brain cannot detect the direction of such low frequency sounds.

FIG-24.22

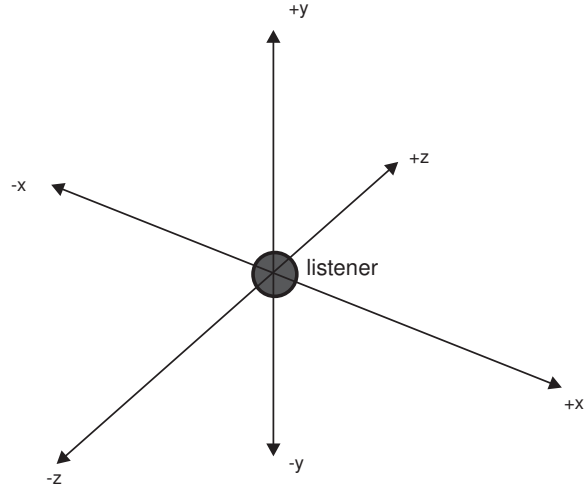
A Suitable Setup for 3D Sound



If we image we're floating in the centre of a room, then sounds can come from our left or right (i.e. anywhere along the x-axis), above or below us (i.e. anywhere along the y-axis), and anywhere in front or behind us (i.e. the z-axis). The concept is shown in FIG-24.23.

FIG-24.23

Our Listening Position in 3D Space



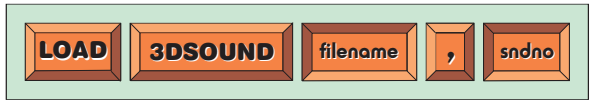
Notice that your head is placed at the origin, where all three axes meet. To our left is the -x part of the x-axis and in front of us is the +z part of the z-axis.

The LOAD 3DSOUND Statement

Although the sounds used are in every way normal mono WAV files, if you intend to create a 3D effect with such a file, it must be loaded using the LOAD 3DSOUND statement which has the following format shown in FIG-24.24.

FIG-24.24

The LOAD 3DSOUND Statement



In the diagram:

filename	is a string specifying the name of the file to be loaded. It is best if the file has been previously copied to your directory using the **Media|Add** option. The file must be a mono WAV file.
sndno	is an integer representing the value by which the sound object created is identified within the program.

A typical example of using this statement would be:

```
LOAD 3DSOUND "help.wav", 1
```

If the requested file is not found or cannot be loaded for some reason, then your program will terminate.

You can play the sound using the normal PLAY SOUND or LOOP SOUND. The sound file will be played at equal volume through all speakers and should give the impression that the sound originates from inside your head.

Activity 24.14

Find out what effects are produced on your own system by executing the following program (*sounds3D.dbpro*):

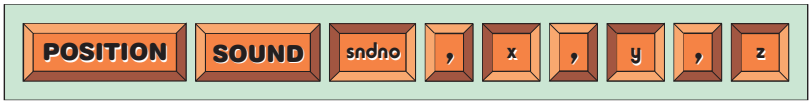
```
LOAD 3DSOUND "laser.wav",1
WAIT 500
LOOP SOUND 1
WAIT KEY
END
```

The POSITION SOUND Statement

We can move the position from which the sound is coming using the POSITION SOUND statement. This statement allows you to specify the 3D coordinates at which the sound is to be placed and has the format shown in FIG-24.25.

FIG-24.25

The POSITION SOUND Statement



In the diagram:

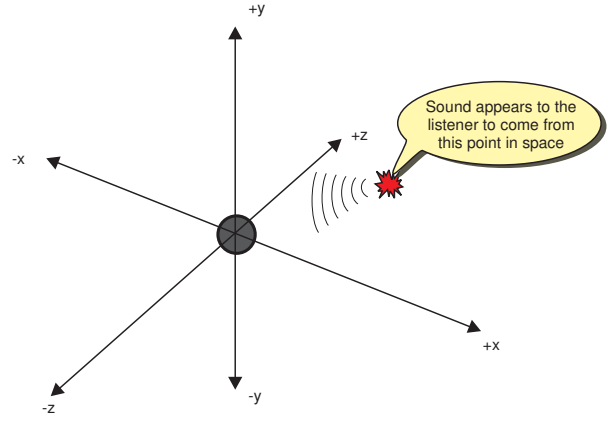
sndno	is the number previously assigned to the 3D sound.
x,y,z	are three real numbers representing the 3D coordinates of the sound's new position. Typically, values for these parameters should be given in 100's.

For example, we could place a sound at position (100,300,50) (see FIG-24.26) using the statement:

```
POSITION SOUND 1,100,300,50
```

FIG-24.26

Positioning a Sound



Activity 24.15

Modify your previous program to add the code

```
FOR x = -300 TO 300 STEP 100
    FOR y = -300 TO 300 STEP 100
        FOR z = -300 TO 300 STEP 100
            POSITION SOUND 1, x, y, z
            WAIT 1000
        NEXT z
    NEXT y
NEXT x
```

This should be placed immediately after the LOOP SOUND statement.

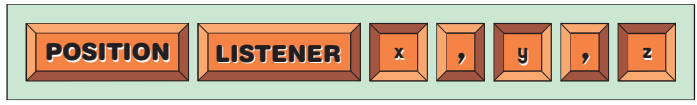
Controlling the Listener

The POSITION LISTENER Statement

Not only can a sound be positioned, but the apparent position of the listener can also be moved. Of course, rather than physically move the listener, the effect is achieved by adjusting the output from the speakers to give the impression that the listener has moved to a new position. The effect is achieved using the POSITION LISTENER statement which has the format shown in FIG-24.27.

FIG-24.27

The POSITION LISTENER Statement



In the diagram:

x,y,z	are three real numbers representing the 3D coordinates to which the listener is to be moved.

For example, we might move the listener forward using the statement:

```
POSITION LISTENER 0,0,200
```

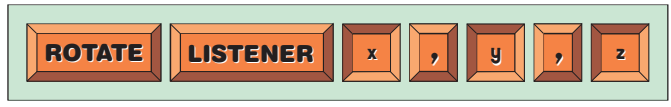
The ROTATE LISTENER Statement

Even though a listener's position is moved, the listener continues to face in a forward direction (towards +z). However, by using the ROTATE LISTENER, we can rotate the listener by 360^{o} around any axis we wish.

Imagine you are sitting in a swivel chair listening to a piece of music. You can turn the chair to the left or right (technically, this is called **rotation about the y-axis**). You can also tilt the chair forwards or backwards - to a limited extent (**rotation about the x-axis**). A final option is to lean your body over to the left or right (**rotation about the z-axis**). The format for this statement is given in FIG-24.28.

FIG-24.28

The ROTATE LISTENER Statement



In the diagram:

x,y,z	are three real numbers representing the degrees of rotation about each axis. These values should all lie in the range 0 to 360.

For example, we can make the listener do the equivalent of swivelling his chair to face backwards with the line

```
ROTATE LISTENER, 0,180,0
```

The SCALE LISTENER Statement

The SCALE LISTENER statement does the equivalent of turning down the listener's hearing sensitivity. In effect this can be used to turn down the volume on all sounds that are playing or to be played. The statement has the format shown in FIG-24.29.

FIG-24.29

The SCALE LISTENER Statement

In the diagram:

factor	is a real value representing the scaling factor. This value should lie between 0.0 and 1.0, with 1.0 being the default, normal value.

Retrieving Data on 3D Sounds and the Listener

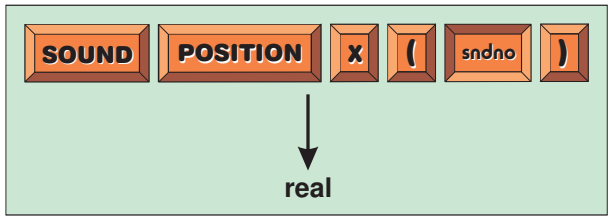
Another batch of statements is available to retrieve the various settings of both 3D sounds and the listener. These are described below.

The SOUND POSITION X Statement

The x ordinate of a 3D sound can be determined using the SOUND POSITION X statement which has the format shown in FIG-24.30.

FIG-24.30

The SOUND POSITION X Statement



In the diagram:

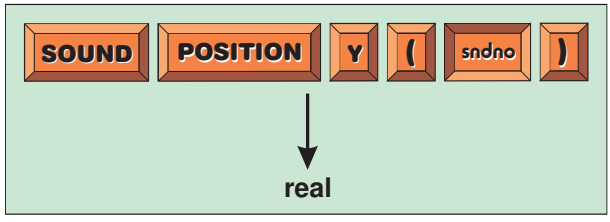
sndno	is the number of the existing 3D sound object whose x ordinate is to be returned.

The SOUND POSITION Y Statement

The y ordinate of a 3D sound can be determined using the SOUND POSITION Y statement which has the format shown in FIG-24.31.

FIG-24.31

The SOUND POSITION Y Statement



In the diagram:

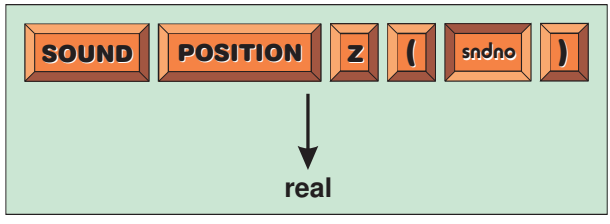
sndno	is the number of the existing 3D sound object whose y ordinate is to be returned.

The SOUND POSITION Z Statement

The z ordinate of a 3D sound can be determined using the SOUND POSITION Z statement which has the format shown in FIG-24.32.

FIG-24.32

The SOUND POSITION Z Statement



In the diagram:

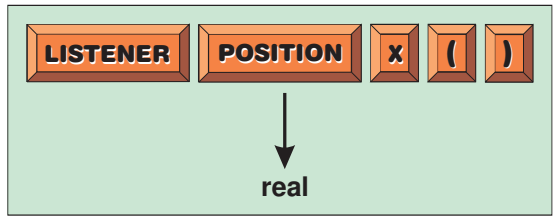
sndno	is the number of the existing 3D sound object whose z ordinate is to be returned.

The LISTENER POSITION X Statement

The x ordinate of the listener can be discovered using the LISTENER POSITION X statement whose format is shown in FIG-24.33.

FIG-24.33

The LISTENER POSITION X Statement

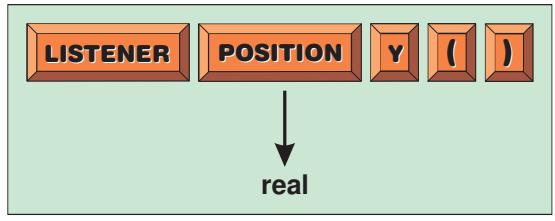


The LISTENER POSITION Y Statement

The y ordinate of the listener can be discovered using the LISTENER POSITION Y statement whose format is shown in FIG-24.34.

FIG-24.34

The LISTENER POSITION Y Statement

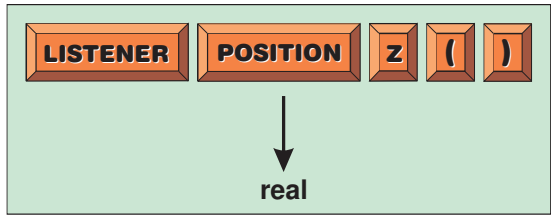


The LISTENER POSITION Z Statement

The z ordinate of the listener can be discovered using the LISTENER POSITION Z statement whose format is shown in FIG-24.35.

FIG-24.35

The LISTENER POSITION Z Statement

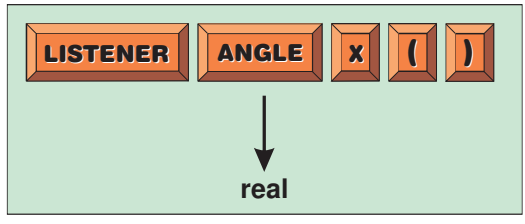


The LISTENER ANGLE X Statement

The angle to which the listener has been rotated about the x-axis (i.e. forward or backward) can be retrieved using the LISTENER ANGLE X statement which has the format shown in FIG-24.36.

FIG-24.36

The LISTENER ANGLE X Statement

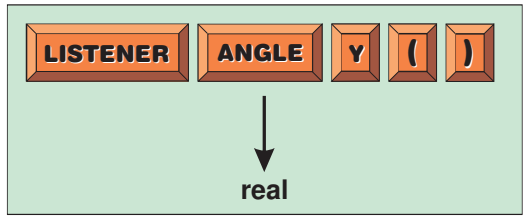


The LISTENER ANGLE Y Statement

The angle to which the listener has been rotated about the y-axis can be retrieved using the LISTENER ANGLE Y statement which has the format shown in FIG-24.37.

FIG-24.37

The LISTENER ANGLE Y Statement

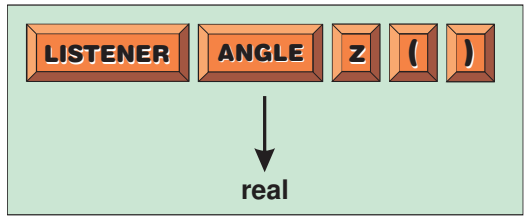


The LISTENER ANGLE Z Statement

The angle to which the listener has been rotated about the z-axis can be retrieved using the LISTENER ANGLE Z statement which has the format shown in FIG-24.38.

FIG-24.38

The LISTENER ANGLE Z Statement



Summary

- A 3D sound can appear to originate from anywhere in space - left, right, in front, behind, below, or above the listener.
- A computer setup needs an appropriate sound card and speaker setup to create 3D sounds.
- The sound file used for 3D sound should be a mono file. The position of the sound within space (that is, the 3D effect) is created by the sound card and speakers.
- Use LOAD 3DSOUND to load a sound file to be used for 3D sound.
- Use POSITION SOUND to specify from where in space a sound should appear to originate.
- Use POSITION LISTENER to place the listener at any point in space.
- Use ROTATE LISTENER to rotate the listener's position.
- Use SCALE LISTENER to adjust the listeners hearing sensitivity.
- SOUND POSITION X returns the x-ordinate of a specified sound's position.
- SOUND POSITION Y returns the y-ordinate of a specified sound's position.
- SOUND POSITION Z returns the z-ordinate of a specified sound's position.
- LISTENER POSITION X returns the x-ordinate of the listener's position.
- LISTENER POSITION Y returns the y-ordinate of the listener's position.
- LISTENER POSITION Z returns the z-ordinate of the listener's position.

- LISTENER ANGLE X returns the listener's angle of rotation about the x-axis.
- LISTENER ANGLE Y returns the listener's angle of rotation about the y-axis.
- LISTENER ANGLE Z returns the listener's angle of rotation about the z-axis.

Solutions

Activity 24.1

No solution required.

Activity 24.2

```
REM *** Load sound ***
LOAD SOUND "welcome.wav", 1
WAIT 500
REM *** Play sound ***
PLAY SOUND 1,20000
REM *** End program ***
WAIT KEY
END
```

Activity 24.3

```
REM *** Load sound ***
LOAD SOUND "welcome.wav", 1
WAIT 500
REM *** Loop sound ***
LOOP SOUND 1,20000,50000
REM *** End program ***
WAIT KEY
END
```

Activity 24.4

The sound resumes playing at the point it was paused but does not loop.

Activity 24.5

```
REM *** Load sound ***
LOAD SOUND "demo.wav", 1
WAIT 500
REM *** Play at half speed ***
SET SOUND SPEED 1,11000
REM *** Play sound continuously ***
LOOP SOUND 1
REM *** Pause when key is pressed ***
WAIT KEY
PAUSE SOUND 1
REM *** Resume when key is pressed ***
WAIT KEY
REM *** Set to double speed ***
SET SOUND SPEED 1,44000
RESUME SOUND 1
REM *** End program ***
WAIT KEY
END
```

Activity 24.6

No solution required.

Activity 24.7

No solution required.

Activity 24.8

```
REM *** Specify file ***
LOAD SOUND "sample.wav",1
REM *** Start recording - 10 seconds ***
PRINT "Speak now "
RECORD SOUND 1,10000
REM *** Stop after 10s or key pressed ***

WAIT KEY
STOP RECORDING SOUND
REM *** Play back the recorded sound ***
PRINT "Replaying now..."
SLEEP 500
PLAY SOUND 1
REM *** Save the sound ***
PRINT "Saving recording to file"
SAVE SOUND "speech.wav",1
REM *** End program ***
WAIT KEY
END
```

A second attempt will fail since the file *speech.wav* already exists this time round.

Activity 24.9

1) The message displayed is:
   SOUND is playing

2) The message displayed is
   SOUND is paused

3) No messages are displayed.

Activity 24.10

```
REM *** Load sound file ***
LOAD SOUND "demo2.wav", 1
REM *** Play sound file ***
PLAY SOUND 1
WAIT KEY
STOP SOUND 1
REM *** Get sound file's status ***
playing = SOUND PLAYING(1)
looping = SOUND LOOPING(1)
paused = SOUND PAUSED(1)
REM *** Display details ***
IF playing = 1
   PRINT "SOUND is playing"
ENDIF
IF looping = 1
   PRINT "SOUND is looping"
ENDIF
IF paused = 1
   PRINT "SOUND is paused"
ENDIF
PRINT "Volume setting : ",SOUND VOLUME(1)
PRINT "Speed setting  : ",SOUND SPEED(1)
REM *** End program ***
WAIT KEY
END
```

Activity 24.11

The changes required are:

Add constants to identify the sound objects:

```
#CONSTANT firstasteroid     51
#CONSTANT ship               1
#CONSTANT firstmissile     101
#CONSTANT launchsound        1
#CONSTANT hitsound           2
```

Create the sound objects in *InitialiseGame()*:

```
FUNCTION InitialiseGame()
   REM *** Set screen resolution ***
   SET DISPLAY MODE 1280, 1024, 32
   REM *** Magenta transparent ***
   SET IMAGE COLORKEY 255,0,255
   REM *** Load ship sprite ***
   LOAD IMAGE "ship.bmp",2
   SPRITE ship, SCREEN WIDTH()/2,
   SCREEN HEIGHT()/2,2
   OFFSET SPRITE ship,
   SPRITE WIDTH(ship)/2,
   SPRITE HEIGHT(ship)/2
   lastfired = TIMER()
   LOAD IMAGE "missilemag.bmp",3
   missileno = firstmissile
   REM *** Set all missile moves to zero ***
   FOR c = 0 TO 9
      missilemoves(c) = 0
   NEXT c
   REM *** Seed random number generator ***
   RANDOMIZE TIMER()
   REM *** Load sound objects ***
   LOAD SOUND "launch.wav",launchsound
   LOAD SOUND "explode.wav",hitsound
   REM *** Create asteroids ***
   CreateAsteroids()
   REM *** position asteroids ***
   StartPositionsForAsteroids()
ENDFUNCTION
```

Play the sound when a missile is launched:

```
FUNCTION LaunchMissile()
   REM *** IF missile exists, exit ***
   IF SPRITE EXIST(missileno)
      EXITFUNCTION
   ENDIF
   REM *** Create a new missile sprite ***
   REM *** Match position with ship ***
   SPRITE missileno, SPRITE X(ship),
   SPRITE Y(ship),3
   OFFSET SPRITE missileno,
   SPRITE WIDTH(missileno)/2,
   SPRITE HEIGHT(missileno)/2
   ROTATE SPRITE missileno,
   SPRITE ANGLE(ship)
   MOVE SPRITE missileno, 30
   REM *** Play launch sound ***
   PLAY SOUND launchsound
   REM *** Record time missile was fired ***
   lastfired = TIMER()
   REM *** Add to missileno ***
   missileno = missileno mod 10 +
firstmissile
ENDFUNCTION
```

Play sound when asteroid is hit:

```
FUNCTION HandleMissiles()
   REM *** FOR each possible missile DO ***
   FOR spriteno = firstmissile TO
   firstmissile + 9
      REM *** IF it exists, THEN ***
      IF SPRITE EXIST(spriteno)
         REM *** Move the sprite ***
         MOVE SPRITE spriteno, 15
         REM *** IF missile hits THEN  ***
         spritehit = SPRITE HIT(spriteno,0)
         IF spritehit > 1
            REM *** Play hit sound ***
            PLAY SOUND hitsound
            REM *** Delete sprites hit ***
            DELETE SPRITE spritehit
            DELETE SPRITE spriteno
            REM *** Zeroise move count ***
            post = spriteno - firstmissile
            missilemoves(post)=0
         ELSE
            REM *** Inc missile moves ***
            post = spriteno - firstmissile
            INC missilemoves(post)
            REM *** IF 40 moves, destroy ***
            IF missilemoves(post) >= 40
               DELETE SPRITE spriteno
               missilemoves(post) = 0
            ENDIF
         ENDIF
      ENDIF
   NEXT spriteno
ENDFUNCTION
```

Activity 24.12

Reversing the FOR loop causes the sound to travel from right to left.

Activity 24.13

No solution required.

Activity 24.14

No solution required.

Activity 24.15

```
LOAD 3DSOUND "laser.wav"
WAIT 500
LOOP SOUND 1
FOR x = -300 TO 300 STEP 100
   FOR y = -300 TO 300 STEP 100
      FOR z = -300 TO 300 STEP 100
         POSITION SOUND 1, x, y, z
         WAIT 1000
      NEXT z
   NEXT y
NEXT x
WAIT KEY
END
```