Merlin

British Telecom Business Systems

BASIC

Tonto personal information centre

Technical Publication TPU 12C November 1984

Item Code no. 983081

If you find any errors in this publication or would like to make suggestions for improvement, then please write to:

The Technical Publications Unit
Room 532
British Telecom Merlin
Anzani House
Trinity Avenue
Felixstowe
IP11 8XB

Telephone: Felixstowe (0394) 693787
Telex: 987062 BTANZ

Printed by ICL Printing Services
Engineering Training Centre
Icknield Way West
Letchworth, Herts SG6 4AS

R51017/01

Contents

This guide has four parts. Each part is divided into sections, which are introduced on the first page of the part.

The quickest way to locate information on a particular topic is to use the comprehensive index at the back of the guide.

Part A Introduction

1 How a computer works

A computer works by taking in information, working on that information according to pre-determined instructions, and giving out the results when its calculations are complete.

A COMPUTER PROGRAM

A computer must be given its instructions in a very precise and detailed way and in the exact order in which you want those instructions carried out. Such a list of instructions is called a **program**. You must also give the computer some information on which to work according to your instructions, this information is called **data**. For example in the following statement

```
PRINT 2 + 2
```

PRINT is the instruction which tells the computer to display a result

2 + 2 is the data supplied to the print instruction.

The result, as you would expect, is 4.

You will see later that data is often written into a program but you can also write programs which prompt you for data as a program is running, in which case you type the data in at the keyboard as required.

WHAT IS BASIC?

Programs have to be written in a language the computer can understand. The TONTO uses the computer language (or **programming language**) called BASIC, the language used by the majority of microcomputers.

Vocabulary

BASIC is quite easy to learn and use, as it has a total vocabulary of only about 100 words, called **keywords**. Many BASIC keywords are based on English words and their meanings are obvious: for example, you have already seen **PRINT** used in the example above; **INPUT** means to put information into a program; **RUN** instructs the computer to execute (i.e. run) a program.

Each keyword used in BASIC is an instruction (otherwise known as a **statement**) which tells the computer to do something. A series of statements and their related data constitute a program.

Syntax

BASIC also has certain rules (equivalent to its own grammar) called **syntax**, which you must obey for the TONTO to be able to run your program. In the Keywords section, the syntax for each statement is given under the heading Format. The syntax for items used in several statement types is given in Appendix 1.

Semantics

Even with correct syntax, your program may not do exactly what you intend it to do. Remember that a computer will always try to follow your instructions even if the instructions are wrong. To write useful programs you need to work out exactly what you want to do and all the steps you need to achieve the result you require.

TONTO BASIC Features and facilities

TONTO BASIC has been designed to be even more user-friendly than many earlier versions of BASIC. It offers advanced program structuring, editing and operating facilities. It also contains several features to simplify programming and can often produce results where other versions of BASIC would fail.

Error handling

However, some programs will inevitably not work straight away in which case you can use the error handling facilities of TONTO BASIC to find your mistakes. Such mistakes are called bugs. They can be such things as misspelt words, letters instead of numbers or vice versa (e.g. I for 1, O for Ø), or missing quotation marks or punctuation. Such errors are reported as syntax errors and you can easily correct them using the editing facilities of TONTO BASIC. Remember to check everything you type in very carefully before entering a line into a program. This will help you to minimise errors.

2 How to use the manual

FOR BEGINNERS

Part B of this manual is the Beginner's Guide which gives you step by step instructions on how to write your own BASIC programs. It introduces you to the computer hardware and what you need to know before you start writing programs. The main BASIC words are then introduced one by one with short program examples to show you how they are used. At the end of each section there are some questions for you to answer so that you can check that you understand each section fully before moving on to the next.

Sections 1 to 7 introduce you to the features and facilities of BASIC, Section 8 defines the differences between TONTO BASIC and other versions of BASIC, while the subsequent sections explain the ideas introduced in the earlier sections more fully and give you examples of more advanced programming techniques.

FOR EXPERIENCED PROGRAMMERS

You may wish to glance through the first seven sections of Part B of this manual which contain the rudiments of BASIC programming. The introduction to Part B gives a brief description of the contents of each section which should help you to select those sections relevant to your level of experience.

If you are familiar with other versions of BASIC you may wish to move straight to section 8 in Part B which defines the differences between TONTO BASIC and other versions of BASIC. Parts C and D of this manual provide detailed reference information on the concepts and keywords respectively, used in TONTO BASIC. Appendix 1 contains the syntax definitions of TONTO BASIC. Appendix 2 lists reserved words in TONTO BASIC. Appendix 3 describes how you can exchange data between BASIC and other applications.

FOR ALL

Although you do not need any previous experience of programming or computers to use this manual, it is assumed that you are familiar with the operation of the TONTO as details of setting up, starting up and keyboard use are not given in this manual. You should also have a copy of the

Handbook

Handy for reference purposes.

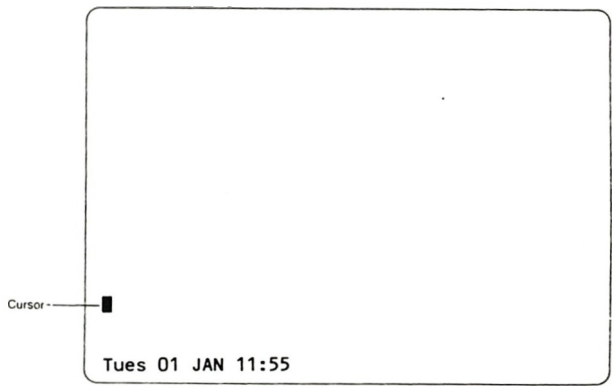
Part B Beginner's Guide

1 Getting started

To use TONTO BASIC you must first load BASIC from either the BASIC cartridge or the WELCOME cartridge in one of your microdrives (see Handbook for details of how to load a cartridge). Press the **START** key to obtain the Top Level Menu, and select BASIC by typing 7, the number of the BASIC option.

THE SCREEN

When the TONTO enters BASIC, the copyright screen appears followed after a few seconds by the start screen as illustrated below:

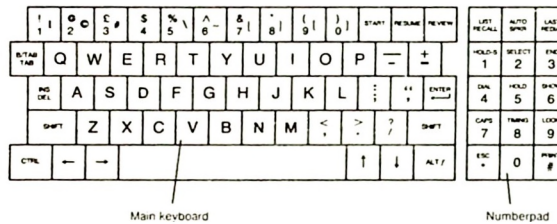


If your screen does not look like this, refer to the problem solving section in the Handbook. This should enable you to resolve any difficulties.

In BASIC different parts of the screen are used for different types of display. Details are given in section 8, Screen Organisation.

THE KEYBOARD

You will already be familar with the TONTO keyboard if you have used the TONTO for other applications before selecting BASIC. Certain facilities of the keyboard which you need to use in BASIC are described below and the whole keyboard layout is shown in the following diagram. Full details of the keyboard are given in the Handbook.



Interrupting BASIC

The **BREAK** key sequence permits you to interrupt BASIC and divert its attention back to you. It may be used when for example:

- you want to stop a program
- you want to abandon changes you are making to a program

The **BREAK** key sequence has been made difficult to type accidentally. To use **BREAK**

hold down CTRL then press SPACE

then release CTRL and SPACE

The TONTO usually responds

not complete

If **BREAK** has been used to interrupt a running program, it can be resumed by typing

CONTINUE ←┘

If you have added or removed any lines from the program, or made any other changes to the program before typing **CONTINUE**, the TONTO replies

Bad line

Type the following:

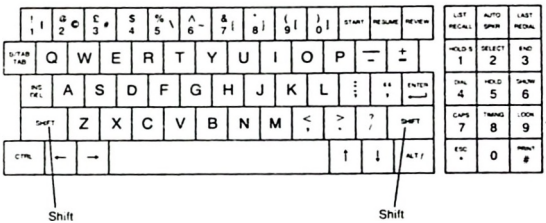
REP A : PRINT INKEY$(-1); : END REP A

then press the key marked **ENTER**

Now everything that you type will appear at the top of the screen. The only way to interrupt this is to use the **BREAK** key sequence. Try it now!

Shift

The **SHIFT** key gives you access to the function or character engraved at the top of a key.



As you can see there are two **SHIFT** keys on the keyboard. You may use whichever is the most convenient.

Hold down one **SHIFT** key and press one of the letter keys. The TONTO displays that letter in capitals (upper case).

Now hold down the **SHIFT** key and press another key, but not a letter, **RETURN**, or one of the numbered keys. The TONTO displays the symbol shown on the upper position of that key (if there is one).

Without a **SHIFT** key you get small (lower case) letters or a symbol in a lower position on the key.

Capitals lock

You can lock capital letters on or off using the CAPS key (SHIFT and number 7 on the numberpad).

CAPS LOCK works like a switch which affects only the letter keys on the keyboard. Press it once, and the unshifted letter keys are locked into a particular mode - upper case or lower case.

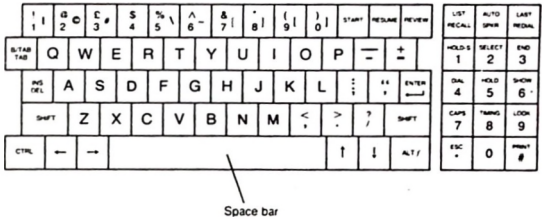
Type some letter keys
Press the **SHIFT** key and hold **SHIFT** down while you press 7 on the numberpad.
Type some letter keys.

You can see that the mode changes and remains until you type the **CAPS LOCK** sequence again.

While **CAPS LOCK** is on, the message CAPS is displayed in the bottom left hand corner of the noticeboard.

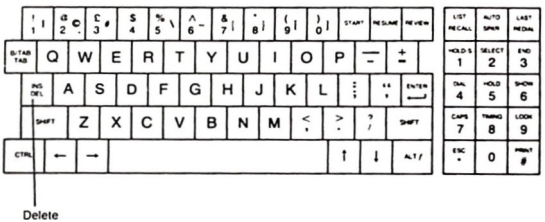
Spacing

The long key at the bottom of the keyboard generates spaces. Use of spaces is very important in BASIC, as you will see in the next section.



Rubbing out

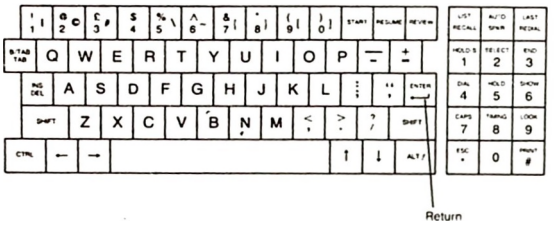
The **DEL** key acts like a rubber. Each time you press it the immediately previous character (if any) is deleted.



Details of **DEL** and the other **editing** keys are given in the Handbook. See also pB2-9 for details of editing.

Entering data

When you have typed a complete message or instruction (for example **RUN**), you must press the **RETURN** key to enter it into the system for action.



Since this key is used so often it is shown by the symbol ←┘. Test the ←┘ key by typing:

```
PRINT "Correct" ←┘
```

If you made no mistakes, the system responds with:

```
Correct
```

Note that the **RETURN** (←┘) and **ENTER** (**SHIFT/RETURN**) keys have equivalent effects except when entering lines in an **AUTO** or **EDIT** command. See pages B5-2 and C2-17 for further details.

Upper and lower case

BASIC recognises keywords whether they are in upper or lower case. For example the BASIC command to clear the screen is **CLS** and can be typed in as:

```
CLS ←┘
cls ←┘
clS ←┘
```

These are all correct and have the same effect. BASIC displays some keywords partly in upper case, where the upper case portion shows the allowed abbreviation in order for that keyword to be recognised by the system. For example, the BASIC keyword **REP**eat may be abbreviated to **REP**. Where a keyword cannot be abbreviated it is displayed completely in upper case.

Use of quotes

The usual use of quotes (" or ') is to define a word or sentence, that is a string of characters. Try:

```
PRINT "This works" ←┘
```

The computer responds with:

```
This works
```

The quotes are not printed on the screen but they indicate that some text is to be printed and they define exactly what it is - everything between the opening and closing quote marks. If you wish to use the quote symbol itself in a string of characters then the apostrophe symbol can be used to define the string instead. For example:

```
PRINT 'The quote symbol is"' ←┘
```

prints

```
The quote symbol is"
```

Common typing errors

Zero and letter 'O'

The zero key is the last of the numeric digits at the top of the keyboard, and the zero symbol is slightly thinner than the letter O. There is also a zero key on the numberpad at the right hand side of the keyboard.

The letter O key is among the other letters. Be careful to use the right symbol.

Similarly, avoid confusion between 1, the first of the numeric digits at the top of the keyboard or on the numberpad, and the letter I among the letters.

Keep shift down

When using a **SHIFT** key you must hold it down, press the other key, and only then release the **SHIFT** key.

Note that the same rule applies to the control **CTRL** and alternate **ALT** keys which are used in conjunction with other keys, but you do not need these at present.

A COMPUTER PROGRAM

Type the two simple instructions:

```
CLS ←┘
PRINT 'Hello' ←┘
```

Strictly speaking these instructions constitute a computer program which is executed as soon as you press ←┘(ENTER) and not stored for future use. However, it is the stored program that is important in computing.

Type the same program with line numbers:

```
10 CLS ←┘
20 PRINT 'HELLO' ←┘
```

This time nothing happens externally except that the program appears in the upper part of the screen. This means that the program is accepted as having correct grammar or **syntax**, that is, it conforms to the rules of BASIC, but it has not yet been executed, merely stored. To make it work, type:

```
RUN ←┘
```

The distinction between direct commands for immediate action and a stored sequence of instructions is discussed in the next chapter. For the present you can experiment with the above ideas and two more:

```
LIST ←┘
```

causes an internally stored program to be displayed (listed) on the screen.

```
NEW ←┘
```

causes an internally stored program to be deleted so that you can type in a new one.

SELF TEST ON SECTION 1

You can score a maximum of 15 points from the following test. Check your score with the answers on the next page.

1 In what circumstances might you use the **BREAK** sequence?

2 Name two differences between a **SHIFT** key and the **CAPS LOCK** sequence

3 How can you delete a wrong character which you have just typed?

4 What is the purpose of the **RETURN** key?

5 What symbol represents the **RETURN** key?

What is the effect of the commands in questions 6 to 9?

6 **CLS** ←┘

7 **RUN** ←┘

8 **LIST** ←┘

9 **NEW** ←┘

10 Do keywords have the proper effect if you type them in lower case?

11 What is the significance of those parts of keywords which BASIC displays in upper case?

ANSWERS TO SELF TEST ON SECTION 1

1 Use the **BREAK** sequence to:

interrupt a running program
abandon what you are typing

(2 points)

2 The **SHIFT** key

a) is only effective while you are holding it down whereas the **CAPS LOCK** sequence stays effective until you repeat it (1 point)

b) affects all the letter, digit and symbol keys, but **CAPS LOCK** affects only letters (1 point)

3 The **DEL** key deletes any previous character just left of the cursor

4 The ←┘(**RETURN**) key causes a message or instruction to be entered for action by the computer

5 We use ←┘ for the **RETURN** key

6 **CLS** causes part or all of the screen to be cleared

7 **RUN** causes a stored program to be executed

8 **LIST** causes a stored program to be displayed on the screen

9 **NEW** clears BASIC's store ready for a new program

10 Yes: BASIC keywords are recognised in upper or lower case

11 The part of a keyword displayed in upper case is the allowed abbreviation

CHECK YOUR SCORE

13 to 15 is very good. Carry on reading.

11 or 12 is good, but re-read some parts of section 1.

9 or 10 is fair, but re-read some parts of section 1 and do the test again.

Under 9. You should work carefully through section 1 again and repeat the test.

2 Instructing the computer

To make the computer do something more useful than just displaying the results of your single commands on the screen, you have to give it information or **data** to work on. The computer keeps this information in its store until you tell it to use it. Data can be numbers or characters (see pB3-1 for how to deal with characters).

NAMES AND PIGEON HOLES FOR NUMBERS

When you put data into BASIC's store you need to be able to find it again. The computer keeps your data in storage areas which can be imagined as pigeon-holes.

Though you cannot see them, you need to label (i.e. to give names to) each pigeon-hole so that you can tell the computer where to find your data. Suppose you want to solve the following problem.

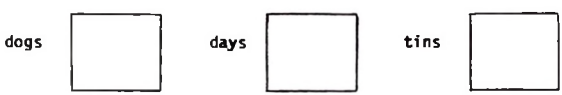
Example

A dog breeder has 9 dogs to feed for 28 days, each at the rate of one tin of 'Beefo' per day. Make the computer print (i.e. display on the screen) the required number of tins.

One way of solving this problem would require three pigeon holes for :

- number of dogs
- number of days
- total number of tins

BASIC allows you to choose any names for pigeon-holes provided you obey certain rules (see page B2-7) and you may choose as shown:



You can make the computer set up a pigeon-hole, name it, and store a number in it with a single instruction or statement such as:

```
LET dogs = 9 ←┘
```

This sets up an internal pigeon-hole, named dogs, and places in it the number 9 thus:

The word **LET** has a special meaning in BASIC. It is called a **keyword**.

BASIC has many other keywords which you will see later. You must be careful to leave a space after **LET** (and all the other BASIC keywords) and the word which follows it. Because BASIC allows you to choose pigeon-hole names with great freedom, LETdogs would be a valid pigeon-hole name.

The **LET** keyword is optional in TONTO BASIC and because of this statements like:

```
LETdogs = 3 ←┘
```

are valid. This would refer to a pigeon-hole called LETdogs.

Names, numbers and keywords should always be separated from each other by spaces if they are not separated by special characters.

Even if it were not necessary, a program line without proper spacing is bad programming style. Machines with small memory size may force programmers into such habits, but this is not a problem with the TONTO.

You can check that your pigeon-hole exists internally by typing:

```
PRINT dogs ←┘
```

The screen should display what is in the pigeon-hole:

```
 9
```

Again, be careful to put a space after **PRINT**.

To solve the original problem, we can write a program which is a sequence of statements. You can now understand the first two:

```
LET dogs = 9 ↵
LET days = 28 ↵
```

These cause two pigeon-holes to be set up, named, and given numbers (i.e. values).

The next instruction multiplies the two numbers and places the result in a new pigeon-hole called tins.

You might expect the statement to be:

```
LET tins = dogs x days
```

but BASIC uses the symbol * for multiply so we write the statement thus:

```
LET tins = dogs * days ↵
```

The system carries out the following tasks:

1 It gets the values, 9 and 28, from the two pigeon holes named dogs and days

2 It multiplies the value 9 by the value 28

3 It sets up a new pigeon-hole named tins

4 It stores the result of the multiplication in the pigeon-hole named tins

This may seem very complicated but you need to understand the stages involved in solving the problem. The effect can be imagined very simply as shown:

The only remaining task is to make the computer print the result, which you do by typing:

```
PRINT tins ↵
```

This causes the output

```
252
```

to be displayed on the screen.

In summary, the program:

```
LET dogs = 9 ←┘
LET days = 28 ←┘
LET tins = dogs * days ←┘
PRINT tins ←┘
```

causes the internal effects best imagined as three named pigeon-holes containing values:

and the output on the screen:

```
252
```

Of course, you could achieve this result more easily with a calculator or a pencil and paper! You could do it quickly with the TONTO by typing:

```
PRINT 9 * 28 ←┘
```

which would give the answer 252 on the screen. However, the ideas we have discussed are the essential starting points of programming in BASIC. They are so essential that they occur in many computer languages and have been given special names.

1 Names such as dogs, days and tins are called **identifiers**

2 A single instruction such as:

```
LET dogs = 9
```

is called a **statement**

3 The arrangement of name and associated pigeon-hole is called a **variable**. The execution of the above statement stores the value 9 in the pigeon-hole named by the identifier dogs

A statement such as:

```
LET dogs = 9
```

is an instruction for a highly dynamic internal process but the printed text is static and it uses the = sign borrowed from mathematics. It is better to think or say (but not type):

```
LET dogs store the value 9
```

and to think of the process having a right to left direction (do not type this):

```
dogs ← 9
```

The use of = in a **LET** statement is not the same as the use of = in mathematics. For example, if another dog turns up you may wish to write:

```
LET dogs = dogs + 1
```

Mathematically, this is impossible but in terms of computer operations it is simple. If the value of *dogs* before the operation was 9, the value after the operation would be 10. Test this by typing:

```
LET dogs = 9 ←┘
PRINT dogs ←┘
LET dogs = dogs + 1 ←┘
PRINT dogs ←┘
```

The output is:

```
9 from the first PRINT statement
10 from the second PRINT statement
```

proving that the final value in the pigeon-hole is as shown:

A good way to understand what is happening to the pigeon-holes (i.e. variables), is to do a dry run. Simply examine each instruction in turn and write down the values which result from it to show how the pigeon-holes are set up, given values, and how they retain their values as the program is executed.

Thus:

```
LET dogs = 9 ←┘
LET days = 28 ←┘
LET tins = dogs * days ←┘
PRINT tins ←┘
```

dogs	days	tins
9		
9	28	
9	28	252
9	28	252

The output is:

```
252
```

You may notice from the examples so far that a variable name has always been used first on the left hand side of a LET statement. Once the variable is set up and has a value, the corresponding variable name can be used on the right hand side of a LET statement.

Suppose you wish to encourage a small child to save money. You might give two bars of chocolate for every pound saved. If you try to compute this as follows:

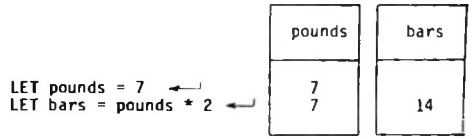
```
LET bars = pounds * 2 ←┘
PRINT bars ←┘
```

you cannot do a dry run as the program stands, because you do not know how many pounds have been saved.

```
LET bars = pounds * 2
```

pounds	bars
?	?

We have made a deliberate error here in using the variable *pounds* on the right of a LET statement without it having been set up and given some value. The TONTO searches internally for the variable *pounds*. It cannot find it, so it concludes that there is an error in the program and gives an error message. We say that the variable *pounds* has not been **initialised** (given an initial value). The program works properly if you do this first:



The program works properly and gives the output:

```
14
```

Identifiers (Names)

You have used names for pigeon holes such as dogs, bars. There are rules about the kinds of names you can choose as follows:

- A name cannot include spaces
- A name must start with a letter
- A name must be made up from letters, digits and _ (underscore)

  allowing you to make names such as:

  dog_food
  month_wage_total

- BASIC does not distinguish between upper and lower case letters, so names like TINS and tins are the same.
- The maximum number of characters in a name is 255.

Names which are constructed according to these rules are called **identifiers**. Identifiers are used for several purposes in BASIC and you need to understand them. The rules allow great freedom in the choice of names so you can make your programs easier to understand. Names like total, count, pens are more helpful than names like Z, P, Q.

A STORED PROGRAM Typing statements without line numbers, as we have mostly done so far, may produce the desired result: there are two reasons however why this method is not satisfactory, except as a first introduction.

1 The program can only execute as fast as you can type; this is not very impressive for a machine with the computing power of the TONTO

2 The individual instructions are not stored after execution so you cannot run the program again, or correct an error, without re-typing the whole thing

Charles Babbage, a nineteenth centry computer pioneer, found that a successful computer needs to store instructions (as well as data) in internal pigeon-holes. These instructions may then be executed rapidly in sequence without further human intervention.

In a stored program each statement starts with a line number. The line number tells BASIC that the statement should be kept. It also determines the position of the statement in the program as statements are executed in ascending sequence. You could use any sequence of line numbers to number your program lines but line numbers usually go up in tens so that you can add extra instructions without renumbering the whole program (see Insert a new line below). When you enter a line preceded by a line number, the system checks the syntax of the line; if correct the line moves to the top of the screen. When you have entered all the program lines, you can execute the program by typing **RUN**. Try this:

```
10 LET price = 15 <-'
20 LET pens = 7 <-'
30 LET cost = price * pens <-'
40 PRINT cost <-'
```

Nothing happens externally yet, but the whole program is stored internally. You make it work by typing:

```
RUN <-'
```

and the output:

```
105
```

appears.

The advantage of this arrangement is that you can edit or add to the program with minimal extra typing.

CHANGING A PROGRAM

In section 5 you will see the full editing features of BASIC, but even at this early stage you can do three things easily:

- replace a line
- insert a new line
- delete a line

Replace a line

Suppose you wish to alter the previous program because the price has changed to 20p for a pen. To change the price you simply retype line 10.

```
10 LET price = 20 ←┘
```

and this line replaces the previous line 10. Assuming the other lines are still stored, test the program by typing:

```
RUN ←┘
```

and the new answer, 140, should appear.

Insert a new line

Suppose you wish to insert a line just before the last one, to print the words 'Total Cost'. This situation often arises so we usually choose line numbers with intervals of 10 (i.e. 10, 20, 30 etc.) to allow space to insert extra lines.

To put in the extra line type:

```
35 PRINT "Total Cost" ←┘
```

and it is inserted between lines 30 and 40. The system allows line numbers in the range 1 to 32767 to allow plenty of flexibility, as it is difficult to be sure in advance what changes may be needed.

Now type:

```
RUN ←┘
```

and the new output is:

```
Total Cost
140
```

Delete a line

You can delete a line by typing the line number, followed by ←┘. For example, to delete line 35 type:

```
35 ←┘
```

The effect is to remove line 35 from your program, such that line 40 now follows line 30.

OUTPUT

The results of what you ask the computer to do are called the **output** from the program. So far you have seen output displayed on the screen as a result of the **PRINT** statements you have used in your programs. As you will see later you can also direct your output to a printer to obtain **hard copy** (the results of your program printed on paper).

The PRINT statement

The **PRINT** statement is very useful. You have seen how to use **PRINT** to display words and numbers on the screen, to print out the contents of variables and to do calculations on the screen. You can **PRINT** text by using quotes or apostrophes:

```
PRINT "chocolate bars" ←┘
```

You can print the values of variables (i.e. contents of pigeon-holes) by typing statements such as:

```
PRINT bars ←┘
```

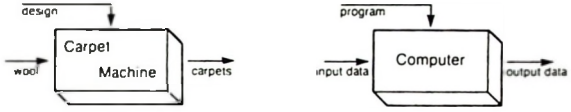
without using quotes.

You will see later how very versatile the **PRINT** statement can be in BASIC. It enables you to place text or other output on the screen exactly where you want it. For now, these two facilities are all you need:

- printing of text
- printing values of variables (contents of pigeon holes)

INPUT

Input collectively describes the instructions (the program) and the information (the data) you give the computer to work on. This can be equated to a manufacturing process. For example, a carpet-making machine needs wool as input. It then makes carpets according to the current design.



If the wool is changed you may get a different carpet.

The same sort of relations exist in a computer, as illustrated in the above diagrams. If you change the input data the output data also changes.

However, when data is input into pigeon holes by means of **LET** there are two disadvantages when you get beyond very trivial programs:

- writing **LET** statements is laborious
- changing such input is also laborious

The INPUT statement

You can arrange for data to be given to a program as it runs. The **INPUT** statement causes the program to pause and wait for you to type in something at the keyboard. First type:

```
NEW ←┘
```

so that the previous stored program (if it is still there) is erased ready for this new one. Now type:

```
 5 CLS ←┘
10 LET price = 15 ←┘
20 PRINT "How many pens?" ←┘
30 INPUT pens ←┘
40 LET cost = price * pens ←┘
50 PRINT cost ←┘
RUN ←┘
```

Note the use of the CLS statement to give the program a clean sheet, by removing the previous display.

The program pauses at line 30 for you to input your data and you should type the number of pens you want, say:

```
4 ←┘
```

Do not forget the **RETURN** key. The output is:

```
 60
```

The **INPUT** statement needs a variable name so that the system knows where to put the data which comes in from your typing at the keyboard. The effect of line 30 with the data you type in is the same as a **LET** statement's effect: the **INPUT** statement is more convenient for some purposes when interaction between computer and user is desirable. However, the **LET** statement and the **INPUT** statement are useful only for modest amounts of data. We need something else to handle larger amounts of data without creating pauses in the execution of the program.

The READ statement

TONTO BASIC, like most BASICs, provides another method of input known as **READ**ing from **DATA** statements. We can re-type the above program in a new form, to give the same effects without any pauses. Try this:

```
NEW ←┘
10 CLS ←┘
20 READ price, pens ←┘
30 LET cost = price * pens ←┘
40 PRINT cost ←┘
50 DATA 15,4 ←┘
RUN ←┘
```

The output is:

```
 60
```

as before.

Try running the program a second time. The message

```
 At line 20 end of file
```

is output. This is explained below under the **RESTORE** statement.

The DATA statement

When line 10 is executed the system searches the program for a **DATA** statement. It then uses the values in the **DATA** statement for the variables in the **READ** statement in exactly the same order. **DATA** statements are used by the program but they are not executed in the sense that every other line is executed in turn. **DATA** statements can go anywhere in a program but they are normally placed at the end, out of the way. Think of them as necessary to, but not really part of, the active program. The rules about **READ** and **DATA** are as follows:

1 All **DATA** statements are considered to be a single sequence of items. So far these items have been numbers but they could be anything that can appear on the right hand side of a **LET** statement

2 Every time a **READ** statement is executed the necessary values are copied from the **DATA** statement into the variables named in the **READ** statement

3 The system keeps track of which items have been so **READ**. If a program attempts to **READ** more items than exist in all the **DATA** statements, an error is signalled

The RESTORE Statement

The first time a program **READ**s a value from a **DATA** statement the value selected is the first value given in the lowest numbered **DATA** statement. The next **READ** selects the next value in that **DATA** statement if there is one, otherwise it selects the first value of the next **DATA** statement, and so on until the end of the program is reached.

The **RESTORE** statement is used to reset the point at which the search for a value begins. Try adding

```
15 RESTORE ←┘
```

to the previous program. If you **RUN** this program now, it will not produce an end of file message but will produce the output

```
60
```

as it did the first time.

RESTORE in this form tells the system to start its search for **DATA** at the beginning of the program. You can start the search from any line by specifying a line number. For example, if you amend the previous program by typing

```
15 RESTORE 50 ←┘
60 DATA 12, 3 ←┘
```

and type **RUN** the output is

```
 36
```

because the search for **DATA** starts at line 60.

SELF TEST ON SECTION 2

You can score a maximum of 21 points from this test. Check your score with the answers on the next page.

1 How would you describe the concept of a pigeon-hole?

2 State two ways of creating an internal pigeon-hole and storing a value in it (2 points)

3 How can you find out the value of an internal pigeon-hole?

4 What is the usual technical name for such a pigeon-hole?

5 When does a pigeon-hole get its first value?

6 A variable is so called because its value can vary as a program is executed. What is the usual way of causing such a change?

7 The = sign in a **LET** statement does not mean 'equals' as in mathematics. What does it mean?

8 What happens when you enter an unnumbered statement?

9 What happens when you enter a numbered statement?

10 What is the purpose of quotes in a **PRINT** statement?

11 What happens when you do not use quotes in a **PRINT** statement?

12 What does an **INPUT** statement do which a **LET** statement does not?

13 What type of program statement is never executed?

14 What is the purpose of a **DATA** statement?

15 What is another word for the name of a pigeon-hole (variable name)?

16 Write down three valid identifiers which use just letters; letters and digits; letters and underscores? (3 points)

17 Why is the space bar especially important in BASIC?

18 Why are freely chosen identifiers important in programming?

ANSWERS TO SELF TEST ON SECTION 2

1 A pigeon-hole is like an internal store which you can name and put values into.

2 A **LET** statement which uses a particular name for the first time causes a pigeon-hole to be created and named, for example

```
LET count = 1
```

(1 point)

A **READ** statement which uses a name for the first time has the same effect, for example

```
READ count
```

(1 point)

3 You can find the value of a pigeon-hole with a **PRINT** statement.

4 The technical name for a pigeon-hole is **variable** because its value can vary as a program runs.

5 A variable gets its first value when it is first used in a **LET** statement, **INPUT** statement or **READ** statement.

6 A change in the value of a variable is usually caused by the execution of a **LET** statement.

7 The = sign in a **LET** statement represents an operation:

'Evaluate whatever is on the right hand side and place it in the pigeon hole named on the left hand side', that is 'Let the left hand side become equal to the right hand side'.

8 An unnumbered statement is executed immediately.

9 A numbered statement is not executed immediately. It is stored, and displayed on the screen.

10 The quotes in a **PRINT** statement enclose text which is to be printed.

11 When quotes are not used, you are printing out the value of a variable.

12 An **INPUT** statement makes the program pause so that you can type data at the keyboard.

13 **DATA** statements are never executed.

14 They are used to provide values for the variables in **READ** statements.

15 The technical word for the name of a pigeon hole is **identifier.**

16 Example answers:

```
day
day__23
day__of__week
```

17 The space bar is especially important for putting spaces after or before keywords so that they cannot be taken as identifiers (names) chosen by the user.

18 Freely chosen identifiers are important because they help you to make programs easier to understand. Such programs are less prone to errors and are more adaptable.

CHECK YOUR SCORE

---

18 to 21 very good. Carry on reading.

16 to 17 good but re-read some parts of section two.

14 to 15 fair, but re-read some parts of section 2 and do the test again.

Under 14 you should work carefully through section 2 again and repeat the test.

EXERCISES ON SECTION 2

1 Carry out a dry run to show the values of all variables as each line of the following program is executed:

```
10 LET hours = 40 ←┘
20 LET rate  = 3  ←┘
30 wage = hours * rate ←┘
40 PRINT hours, rate, wage ←┘
```

2 Write and test a program, similar to that of exercise 1, which computes the area of a carpet 3 metres in width and 4 metres in length. Use the variable names: width, length, area

3 Re-write the program of exercise 1 so that it uses two **INPUT** statements instead of **LET** statements.

4 Re-write the program of exercise 1 so that the input data (40 and 3) appears in a **DATA** statement instead of a **LET** statement.

5 Re-write the solution to exercise 2 using a different method of data input. Use **READ** and **DATA** if you originally used **LET** and vice-versa.

6 Bill and Ben always argue about who is poorer, so they decide to have a gamble. Each takes all the pound notes out of his wallet and gives them to the other. Write a program to simulate this entirely with **LET** and **PRINT** statements. Use a third person, Sue, to hold Bill's money while he accepts Ben's.

7 Re-write the program of exercise 6 so that a **DATA** statement holds the two numbers to be exchanged.

3 Characters and strings

Teachers sometimes wish to assess the reading ability needed for particular books or classroom materials. Various tests are used and some of these compute the average lengths of words and sentences. We will introduce ideas about handling words or character strings by examining simple ways of finding average word lengths.

A character string is a sequence of letters, digits or other symbols all of which may or may not be words. That is why the term **character string** is used. It is usually abbreviated to **string**. Strings are handled in ways similar to number handling but, of course, we do not do the same operations on them. We do not multiply or subtract strings. We join them, separate them, search them and generally manipulate them as we need.

NAMES AND PIGEON HOLES FOR STRINGS

You can put character strings into variables and use the information just as you do with numbers but you must use a different kind of label or name for a variable which contains strings. If you intend to store (not all at once) words such as:

FIRST SECOND THIRD
and
JANUARY FEBRUARY MARCH

you may choose to name two variables:

Notice the dollar sign. Variables for strings are internally different from those for numbers and BASIC needs to know which is which. All names of variables for strings must end with **$**. Otherwise the rules for choosing names are the same as the rules for the names of numeric variables.

You can pronounce:

day_number$ as daynumberstring
month$ as monthstring

The LET statement works in the same way as for numbers except that you must enclose the letters or symbols used in the string in quote marks. If you type:

```
LET day_number$ = "FIRST" ←┘
```

an internal pigeon hole, named *day_number$*, will be set up with FIRST in it thus:

The quote marks are not stored. They are used in the LET statement to make it absolutely clear what is to be stored in the variable. You can use a pair of apostrophes instead of a pair of quote marks. You can check by typing:

```
PRINT day_number$ ←┘
```

and the screen should display what is in the variable:

```
FIRST
```

Identifiers and String Variables

Names of variables, such as:

day_number$
word$
month$
phrases$

are called **string identifiers**. The dollar symbol means that the variables may store character strings. The dollar symbol must always be at the end of the string identifier.

Variables of this kind are called **string variables** because they contain only character strings which may vary as a program runs.

The contents of string variables, like the contents of other variables, are called values. Thus words like 'FIRST' and 'OF' might be values of string variables named *day_number$* and word$.

Lengths of strings

BASIC makes it easy to find the length of, or number of characters in, any string. You simply write, for example:

```
PRINT LEN(month$) ←┘
```

If the variable, *month$*, contains SEPTEMBER the number 9 is displayed. You can see the effect in a simple program:

```
NEW ←┘
10 LET month$ = "SEPTEMBER" ←┘
20 PRINT LEN(month$) ←┘
RUN ←┘
```

The screen displays:

```
 9
```

LEN is a BASIC keyword.

An alternative method of achieving the same result uses both a string variable and a numeric variable.

```
NEW ←┘
10 LET month$ = "SEPTEMBER" ←┘
20 LET length = LEN(month$) ←┘
30 PRINT length ←┘
RUN ←┘
```

The screen displays:

```
 9
```

as before, and two internal pigeon-holes contain the values shown:

Program design

Suppose you wanted to write a program to find the average length of the three words:

```
FIRST, OF, FEBRUARY
```

When problems become more complex, it is a good idea to construct a **program design** before writing the program itself. An example for the above program would be:

1 Store the three words in pigeon holes

2 Compute the lengths and store them

3 Compute the average

4 Print the result

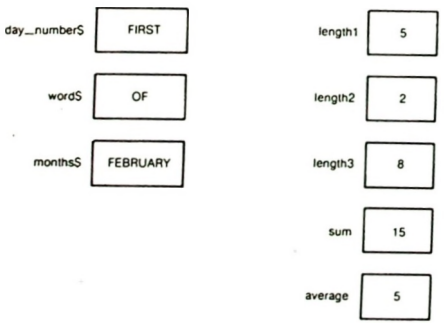
Program

```
NEW ↵
10 LET day_number$ = "FIRST" ↵
20 LET word$ = "OF" ↵
30 LET month$ = "FEBRUARY" ↵
40 LET length1 = LEN(day_number$) ↵
50 LET length2 = LEN(word$) ↵
60 LET length3 = LEN(month$) ↵
70 LET SUM = length1 + length2 + length3 ↵
80 LET average = sum / 3 ↵
90 PRINT average ↵
RUN ↵
```

The symbol / means divided by. The output or result of running the program is simply:

```
 5
```

and there are eight internal pigeon holes involved:



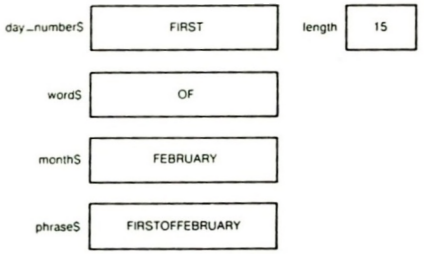
If you think that is a lot of fuss for a fairly simple problem you can easily shorten the program. The shortest version would be a single line but it would be less easy to read. A reasonable compromise uses the symbol & which stands for the operation:

Join two strings

Now type:

```
NEW ←┘
10 LET day_number$ = "FIRST" ←┘
20 LET word$ = "OF" ←┘
30 LET month$ = "FEBRUARY" ←┘
40 LET phrase$ = day_number$ & word$ & month$ ←┘
50 LET length = LEN(phrase$) ←┘
60 PRINT length / 3 ←┘
RUN ←┘
```

The output is 5 as before but there are some different internal effects:



There is one more reasonable simplification which is to use **READ** and **DATA** instead of the first three LET statements. Type:

```
NEW ←┘
10 READ day_number$, word$, month$ ←┘
20 LET phrase$ = day_number$ & word$ & month$ ←┘
30 LET length = LEN(phrase$) ←┘
40 PRINT length / 3 ←┘
50 DATA "FIRST", "OF", "FEBRUARY" ←┘
RUN ←┘
```

The internal effects of this version are exactly the same as those of the previous one. **READ** causes the setting up of internal pigeon holes with values in them, in a similar way to **LET**.

CHARACTERS AND NUMBERS

You have seen that TONTO BASIC can store characters as well as numbers in its pigeon-holes. Although you might never realise, TONTO BASIC actually converts all characters into numbers before storing them. These numbers are called **character codes.**

For some advanced program you may want to take advantage of this. TONTO BASIC provides two keywords to handle this conversion:

CHR$ converts a character code into its respective character. Try typing:

```
PRINT CHR$(66) ←┘
```

and check that a **B** is displayed.

CODE converts a character into its character code. Try typing:

```
PRINT CODE ("B") ←┘
```

and check that 66 is displayed.

Note that **CHR$** ends with $ as it has a character value while **CODE**, which has a numeric value, does not.

SELF TEST ON SECTION 3

You can score a maximum of 10 points from the following test. Check your score with the answers on the next page.

1 What is a character string?

2 What is the usual abbreviation of the term character string?

3 What distinguishes the name of a string variable?

4 How do some people pronounce a word such as 'word$'?

5 What keyword is used to find the number of characters in a string?

6 What symbol is used to join two strings?

7 Spaces can be part of a string. How are the limits of a string defined?

8 When a statement such as:

```
LET meat$ = "steak"
```

is executed, are the quotes stored?

9 What function will turn a suitable code number into a letter?

10 What function will convert a letter into a character code?

ANSWERS TO SELF TEST ON SECTION 3

1 A character string is a sequence of characters (letters, digits or other symbols).

2 The term character string is often abbreviated to string.

3 A string variable name always ends with $.

4 Names such as word$ are sometimes pronounced wordstring or occasionally worddollar.

5 The keyword **LEN** will find the length or number of characters in a string. For example, if the variable meat$ has the value 'steak' then the statement:

```
PRINT LEN(meat$)
```

outputs 5.

6 The symbol for joining two strings is **&**.

7 The limits of a string are defined by quotes or apostrophes.

8 The quotes are not part of the actual string and are not stored.

9 The function is **CHR$**. You must use it with brackets as in **CHR$(66)**.

10 The function is **CODE**.

CHECK YOUR SCORE

9 or 10 is very good. Carry on reading.

7 or 8 is good but re-read some parts of section 3.

5 or 6 is fair but re-read some parts of section 3, and do the test again.

Under 5, you should work carefully through section 3 again and repeat the test.

EXERCISES ON SECTION 3

1 Store the words 'Good' and 'day' in two separate variables. Use a LET statement to join the values of the two variables in a third variable. Print the result

2 Store the following words in four separate pigeon holes:

light let be there

Join the words to make a sentence adding spaces and a full stop. Store the whole sentence in a variable, *sent$*, and print the sentence and the total number of characters it contains

3 Write a cipher program which encodes characters by shifting up the alphabet (e.g. for A print B, for B print C etc)

4 Loops and decisions

Thus far we have introduced the concept of a program as a set of instructions executed in ascending order of line numbers. While correct, this is very limiting for many applications.

Just imagine how tedious life would be if you had to say to a builder, "Lay a paving stone there, and another there, and another there..." instead of "Lay a path from here to the house"!

LOOPING

The concept of repetition or **looping** is fundamental to programming and is fortunately very simple to grasp.

There are only two points to note with loops:

1 An action is repeated a number of times

2 There is usually at least one condition for stopping the repetition or **exiting** the loop

For example, referring back to the builder,

the action is: "Lay a paving stone",

the terminating conditions are: "When the path reaches the house"

or: "When there are no more paving stones"

or: "When it's 5 o'clock"

The GOTO statement

As TONTO BASIC executes statements in sequence, we have to change this sequence to allow looping. One way to do this is to use the **GOTO** statement.

Very simply, this statement causes TONTO BASIC to continue executing instructions from a specified statement. This program illustrates how it works:

```
NEW ←┘
10 CLS ←┘
20 PRINT "LINE 20" ←┘
30 PRINT "LINE 30" ←┘
40 PRINT "LINE 40" ←┘
RUN ←┘
```

Note the output from this program, then add line:

```
25 GOTO 40 ←┘
```

and run the program again.

As you can see, **LINE 30** no longer appears on the screen. This is because TONTO BASIC has executed lines 10, 20 and 25 then 'jumped' past line 30 to line 40.

By use of **GOTO** statements, BASIC can be made to execute statements in virtually any order, however TONTO BASIC provides clearer ways of doing the same thing.

Structured Loops

The idea of a structured loop is the concept that was discussed earlier in this section.

The action that is to be repeated is enclosed by two statements:

REPEAT and **END REPEAT**

For example:

```
NEW ←┘
10 CLS ←┘
20 REPEAT Today ←┘
30 PRINT "LAY A PAVING STONE" ←┘
40 END REPEAT Today ←┘
50 PRINT "TIME TO STOP" ←┘
RUN ←┘
```

Additionally you will notice *Today* after the **REPEAT** and **END REPEAT** keywords. This name is a special form of **identifier** known as a **loop identifier**. It is used to help TONTO BASIC relate the **REPEAT** to the **END REPEAT** in case there is more than one loop in the program.

Terminating a Loop

If you run the above program you will find that it continually prints **LAY A PAVING STONE**. The only way to stop it is to use the **BREAK** key sequence described earlier.

Such an **infinite loop** is, for most practical purposes, marginally less useful than no loop at all.

(A digression: very occasionally it can be useful to have a loop in which the terminating condition is the **BREAK** key sequence but this is not a good programming practice!)

It is very simple to terminate the loop; just add the line:

```
35 EXIT TODAY ←┘
RUN ←┘
```

Now the program outputs

```
LAY A PAVING STONE
TIME TO STOP
```

just as if lines 20, 35 and 40 were not there.

Conditional statements

Although it may appear that you have gone full circle, you have actually programmed a simple loop with the simplest of terminating conditions:

Terminate the loop after executing it once

The true power of a loop becomes apparent when the terminating condition is not fixed.

TONTO BASIC provides a simple way of making choices using a construct called the **IF** statement.

As an example, add or modify lines 11, 12, 30, 31, 35, 50 and 60 to the previous example to produce the following program:

```
10 CLS ←┘
11 INPUT "How long is the path to be"; Length ←┘
12 INPUT "How long is a paving stone"; Stone ←┘
20 REPEAT Today ←┘
25 IF Length < Stone THEN EXIT Today ←┘
30 PRINT CHR$(0); ←┘
31 LET Length = Length - Stone ←┘
40 END REPEAT Today ←┘
50 IF Length = 0 THEN PRINT \ "Path laid" ←┘
60 IF Length <> 0 THEN PRINT \ "Gap at end" ←┘
```

Now **RUN** the program with various values. You will see that a simulation of the paving stones is displayed whose length varies according to your input values.

Lines 25 and 31 are the key to this program:

25: Determines whether there is space for another paving stone to be laid

31: Calculates the length of the unpaved path

Note: Many loops of the type just used can be replaced by mathematical formulae. This is not in general a good practice as it is always best to reflect the structure of the problem in the program.

If you look closely at line 25 you can see that it consists of four parts:

IF condition **THEN** required action

much as in the English:

IF it is raining **THEN** take an umbrella

Although the interpretation of raining can be subjective, a TONTO BASIC condition can yield only one of **TRUE** or **FALSE**.

Lines 50 and 60 are again **conditional statements** which check to see if a cutdown paving stone is required at the end of the path.

It is interesting to note that only one of these conditions can be true. TONTO BASIC provides a clearer way of writing this as:

```
50 IF Length = 0 THEN ←┘
54    PRINT \"Path laid" ←┘
58 ELSE ←┘
60    PRINT \"Gap at end" ←┘
70 END IF ←┘
```

but this is described more fully later on.

KNOWN GOOD PRACTICE

You have already begun to work effectively with short programs. You may have found the following practices are helpful:

- Use of lower case for identifiers, names of variables (pigeon holes), or REPeat structures etc., so that they are easily distinguishable from keywords
- Indenting of statements to show the content of a repeat structure
- Well-chosen identifiers reflecting what a variable or repeat structure is used for
- Editing a program by:

  replacing a line
  inserting a line
  deleting a line

USING PROGRAMS AS EXAMPLES

You have reached the stage where it is helpful to be able to study programs to learn from them and to try to understand what they do. The mechanics of actually running them should now be well understood and in the following chapters we will dispense with the constant repetition of:

NEW before each program
↵ at the end of each line
RUN to start each program

These must still be used when you wish to enter and run a program. But their omission in the text enables you to see the other details more clearly as you try to imagine what the program does when it runs.

For example, the following program generates random upper case letters until a Z appears. It does not show the words **NEW** or **RUN** or the **RETURN** symbol but you still need to use them.

```
10 REPeat letters
20   num = RND(65 TO 90)
30   cap$ = CHR$(num)
40   PRINT cap$
50   IF cap$ = "Z" THEN EXIT letters
60 END REPeat letters
```

In this and subsequent sections, programs and direct commands are shown without the **RETURN** symbol, but you must use this key as usual. You must also remember to use **NEW** and **RUN** as necessary.

USEFUL PROGRAMMING TECHNIQUES

This section describes a few simple techniques which will help to make your programs easier to understand, write and change.

The REMARK statement

Program lines which start with **REM** are ignored by the computer and are solely for the convenience of the programmer or anyone who may read the program. **REM** is short for *remark* and **REMark** statements are useful for inserting notes into the program to remind you what the program is doing. You can use a **REMark** statement to name a program or to insert any additional notes. **REMark** statements have no effect on the program when it is running.

Automatic line numbering

It is tedious to enter line numbers manually. Instead you can type:

```
AUTO
```

and the TONTO replies with a line number:

```
100
```

If you now type in a program line and press **RETURN**, the line you have typed appears at the top of the screen and the TONTO prompts for the next line by outputting 110 on the screen.

If the line already exists, it will be displayed and you can edit it.

To finish the automatic production of line numbers use the **BREAK** sequence:

> Hold down the **CTRL** and press the **SPACE** bar. This produces a message of the form: line number, not complete. For example,
>
> ```
> 130 not complete
> ```

and line 130 is not included in your program.

Alternatively you can use **ENTER** (**SHIFT** and **RETURN**) to send the last line of input. This has the same effect as **RETURN** but also terminates the **AUTO** command.

If you make a mistake you can continue and **EDIT** the line later.

If you want to start at some particular line number, say 600, and use an increment other than 10, you can type for example:

```
AUTO 600,5
```

for an increment of 5.

Lines are then numbered 600, 605, 610, etc.

Editing a line

To edit a line simply type **EDIT** followed by the line number, for example:

```
EDIT 110
```

The line is then displayed with the cursor at the end thus:

```
110 PRINT "First"
```

You can move the cursor using the cursor control keys:

→ one place right
← one place left

To delete a character to the left of the cursor press **DEL**. This deletes the character before the cursor and then moves the cursor over this space so you can type in a different character, if desired.

To delete the character in the cursor position press the **REMOVE** key sequence:

```
CTRL/DEL
```

This deletes the character under the cursor and closes up the space.

To insert a character into a line, position the cursor where you want the extra character and press

```
INS (SHIFT/DEL)
```

This moves all characters under or to the right of the cursor, one position to the right. This lets you insert another character.

If you wish to edit program lines above or below the current line, use the ↑ or ↓ cursor control keys to move the cursor up or down respectively. You can then edit the new line using the keys as above.

Further details of the cursor control keys and more advanced editing facilities are given in the Concept Reference Guide under Data Entry.

NAMING AND SAVING PROGRAMS

When you write and enter programs of more than a few lines you might want to save them on a microdrive cartridge for future use.

Using Microdrives

Before you use a new Microdrive cartridge it must be formatted. Follow the instructions in the Handbook. The choice of name for the cartridge follows the same rules as for BASIC identifiers but is limited to only 8 characters. It is a good idea to write the name on one of the supplied sticky labels and attach this to the cartridge.

WARNING

If you format a cartridge which holds programs and/or data, ALL the programs and/or data will be lost.

Saving programs

When you have typed in a complete program and entered it into the computer, for example the *path* program on page B4-3, you can save it on a microdrive by inserting a cartridge in the left hand drive and typing:

```
SAVE mdv1_path
```

The program is saved in a Microdrive file called *path*. You may choose a Microdrive file name for your program provided that the name is no more than 12 characters long and follows the rules for a BASIC identifier. There must not be a file of that name on the cartridge already.

Deleting programs

The program named above can be deleted by:

```
DELETE mdv1_path
```

Loading programs

You can load your program again from the left hand drive by typing:

```
LOAD mdv1_path
```

You can run the program after it has been loaded by typing

```
RUN
```

Instead of using **LOAD** followed by **RUN** you can combine the two operations in one command:

```
LRUN MDV1_path
```

The program loads and executes immediately after loading.

So far you have used **MDV1_** which refers to the left hand microdrive. If you want to use the right hand drive use **MDV2_** in place of **MDV1_**

Examining programs

To check a program after it has been loaded type:

```
LIST
```

and the whole program is listed on the screen. If you wish to look at only part of the program, say lines 10 to 50, type:

```
LIST 10 to 50
```

The display can be stopped, and subsequently restarted, at any time by using the space bar; or abandoned using the **BREAK** sequence.

Merging programs

Suppose that you have two programs saved on microdrive 1 as prog1 and prog2 where:

prog1 consists of 10 **PRINT** "First" and prog2 consists of 20 **PRINT** "Second"

If you type:

```
LOAD mdv1_prog1
```

followed by

```
MERGE mdv1_prog2
```

the two programs are merged into one. To verify this, type **LIST** and you should see:

```
10 PRINT "First"
20 PRINT "Second"
```

If you **MERGE** a program, make sure that all its line numbers are different from those of the program already in BASIC's store. Otherwise it will overwrite some of the lines of the first program.

This facility becomes very valuable as you become proficient in handling procedures. It is then quite natural to build up a program by adding procedures or functions to it.

FILES

You have learned how to use microdrives for storing programs and the commands **LOAD** and **SAVE**. You can use cartridges for storing files as well as programs. The word file usually means a sequence of data records, where a record is some set of related information such as name, address and telephone number.

The simplest possible type of file is just a sequence of numbers. To illustrate the idea we will place the numbers 1 to 100 in a file called numbers. However, to be recognised by the system the complete specification must be made up of two parts:

device name	where device name is Microdrive 1 or 2
file name etc.	appended information

For example to create the file, numbers, on a cartridge in Microdrive 1. The device name is:

```
MDV1_
```

and the appended information is just the file name:

```
numbers
```

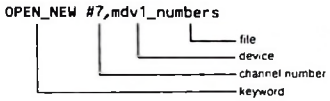
So the complete specification is:

```
MDV1_numbers
```

File specifications have a maximum length of 12 characters.

Channels

It is possible for a program to use several files at once, each referred to by an associated channel number. This can be any integer in the range 0 to 15. A file is associated with a channel number by using the **OPEN** statement or, if it is a new file, **OPEN NEW**. For example you may choose channel 7 for the numbers file and write:



You can now refer to the file just by quoting the number #7.

The complete program is:

```
10 REMark Simple file
20 OPEN_NEW #7, MDV1_numbers
30 FOR number = 1 TO 100
40   PRINT #7, number
50 END FOR number
60 CLOSE #7
```

Use of the **FOR** and **END FOR** keywords are explained in the next section.

The **PRINT** statement causes the numbers to be output to the cartridge file because #7 has been associated with it. The **CLOSE #7** statement is necessary because the system has some internal work to do when the file has been used. It also releases channel 7 for other possible uses.

You need to know that the file is correct and you can only be certain of this if the file is read and checked. The necessary keyword is **OPEN_IN**, otherwise the program for reading data from a file is similar to the previous one.

```
10 REMark Reading a file
20 OPEN_IN #6, MDV1_numbers
30 FOR Item = 1 TO 100
40   INPUT #6, number
50   PRINT ! number !
60 END FOR Item
70 CLOSE #6
```

See page B12-1 for details of ! used as a print separator.

The program should output the numbers 1 to 100, but only if the cartridge containing the file numbers is still in Microdrive **MDV1**.

Devices and channels

A file of data on a microdrive is one example of a device. We say that a file has been opened, although strictly we mean that a device has been associated with a particular channel. Any further necessary information has also been provided. Certain devices have channels associated with them by TONTO BASIC as below:

channel	usual use
#0	entering and editing BASIC commands
#1	**PRINT**ing and **INPUT**ting of data
#2	**LIST**ing of programs

GENERAL

Be careful and methodical with cartridges. Always keep one back-up copy and if you suspect any problem with a cartridge or microdrive keep a second back-up copy. Computer professionals very rarely lose data or programs. They know that even the best machines or devices have occasional faults and they allow for this.

SELF TEST ON SECTION 5

You can score a maximum of 14 points from the following test. Check your score with the answers on the following page.

1 Why are lower case letters preferred for the program words you choose?

2 What is the purpose of indenting?

3 What should normally guide your choice of identifiers for variables and loops?

4 Name three ways of editing a program in the computer's main memory (3 points)

5 What should you remember to type at the end of every command or program line to enter it?

6 What should you normally type before you enter a program at the keyboard?

7 What must be at the beginning of every line to be stored as part of a program?

8 What must you remember to type to make a program execute?

9 What keyword enables you to put information into a program yet has no effect on the execution?

10 Which keyword allows you to store a program on a cartridge?

11 What is the difference between the **LOAD** AND **LRUN** keywords? (2 points)

ANSWERS TO SELF TEST ON SECTION 5

1 Lower case letters for variable names (or loop names) contrast with the keywords which are at least partly displayed in upper case

2 Indenting clarifies the content of loops (and other structures)

3 Identifiers (names) should normally be chosen so that they mean something, for example, count or word$ rather than C or WS

4 You can edit a stored program by:

replacing a line
inserting a line
deleting a line (3 points)

5 The **RETURN** key should be used to enter a command or program line

6 **NEW** ←┘. This wipes out any previous BASIC program in the TONTO and ensures that a new program which you enter will not be merged with an old one

7 If you wish a line to be stored as part of a program then you must use a line number

8 **RUN** followed by ←┘will cause a program to execute

9 **REM**ark

10 The keyword **SAVE** enables programs to be stored on cartridges for subsequent retrieval

11 **LOAD** and **LRUN** both retrieve a program from cartridge, but **LRUN** tells BASIC to run it (2 points)

CHECK YOUR SCORE

12 to 14 is very good. Carry on reading

10 or 11 is good but re-read some parts of section 5

8 or 9 is fair but re-read some parts of section 5 and do the test again

Under 8, you should re-read section 5 carefully and do the test again

---

EXERCISES ON SECTION 5

1 Re-write the following program using lower case letters to give a better presentation. Add the words **NEW** and **RUN**. Use line numbers and the ←┘ symbol just as you would to enter and run a program. Use **REM**ark to give the program a name.

```
LET two$ = "TWO"
LET four$ = "FOUR"
LET six$ = two$ & four$
PRINT LEN(six$)
```

Explain how two and four can produce 7

2 Use indenting, lower case letters, **NEW**, **RUN**, line numbers and the ←┘ symbol to show how you would actually enter and run the following program:

```
REPeat loop
      letter_code = RND(65 TO 90)
  LET letter$ = CHR$(letter_code)
  PRINT letter$
  IF letter$ = 'Z' THEN EXIT loop
END REPeat loop
```

3 Rewrite the following program in better style, using meaningful variable names and good presentation. Write the program as you would enter it:

```
10  LET s = 0
20  REPeat total
30  LET n = RND(1 TO 6)
40  PRINT n !
50  LET s = s + n
60  If n = 6 THEN EXIT total
70  END REPeat total
80  PRINT s
```

Decide what the program does and then enter and run it to check your decision

6 Arrays and FOR loops

WHAT IS AN ARRAY

You know that numbers or character strings can become values of variables. You can picture this as numbers or words going into internal pigeon holes.

Suppose, for example, you want to collect some statistics about your finances over the last few months. Given that you know how much you were paid each month, and all the bills you have settled, you should be able to calculate how rich you're getting each month.

We can assign pigeon-holes, as before, in a number of ways.

Using **LET** statements:

```
10 LET pay_jan   = 987.32
20 LET bills_jan = 323.32
30 LET pay_feb   = 942.27
40 LET bills_feb = 876.11
50 LET pay_mar   = 966.39
60 LET bills_mar = 892.15
70 PRINT "surplus="; pay_jan - bills_jan
80 PRINT "surplus="; pay_feb - bills_feb
90 PRINT "surplus="; pay_mar - bills_mar
```

Using **READ** statements:

```
10 READ pay_jan, pay_feb, pay_mar
20 READ bills_jan, bills_feb, bills_mar
30 PRINT "surplus="; pay_jan - bills_jan
40 PRINT "surplus="; pay_feb - bills_feb
50 PRINT "surplus="; pay_mar - bills_mar
60 DATA 987.32, 942.27, 966.39, 323.32, 876.11 892.15
```

As the quantity of data becomes large the advantages of **READ** and **DATA** over **LET** are apparent, but if you wanted to calculate your surplus pay over the last five years you would still have a lot of typing to do!

The solution to this problem lies in revising the concept of the data items. As opposed to having 6 data items, you actually only have two:

pay and bills

which you might write on paper as:

month	pay	bills
jan	987.23	323.32
feb	942.27	876.11
mar	966.39	892.15

Just as you might have drawn a table, so can BASIC although, for historic reasons, it is called an **array**.

Anything can be stored in an **array** provided that all items are of the same type.

The DIM statement

In order to tell BASIC that you want to use a table or **array** it is necessary to introduce a new statement called the **DIM** statement, which tells BASIC the dimensions of the table.

You will probably decide that your table is 3 columns by 3 rows, but pause a moment -

are bills really the same as pay?

Although they are both measured in the same currency, the two are not of the same logical type, i.e. a bill is a liability, your pay an asset.

Similarly, the month is not the same, and furthermore is not even measured in numeric terms:

To tell BASIC to set up these tables you type:

```
DIM pay(3), bills (3), month$(3,10)
```

where the 3 indicates that there are to be three rows (one for each month) in the table and the 10 means that the maximum number of characters of a data item is ten.

Note that month$ is dimensioned differently. This is because it is a **string array**. These are discussed a little later.

Using an element of an array is simple. Instead of writing:

```
pay_feb
```

you now write:

```
pay(2)
```

so you could rewrite the previous program using arrays.

Indexing arrays

So far you have seen how to set up an array and refer to any element. It may not seem very useful, but consider the following program:

```
10 LET pay(2) = 44
20 LET month  = 2
30 PRINT pay(month)
```

You will be familiar with lines 10 and 20 from previous sections but line 30 introduces a new concept of **array indexing**.

When BASIC looks at this statement, it works in the following way:

- Firstly, BASIC finds **PRINT** and decides that it must print something
- Secondly, BASIC finds pay and decides that it must print an element of this table.
- Thirdly, BASIC replaces month by the value of the variable month which is 2.
- Finally, BASIC prints the value of array element pay(2).

If you think about this, you will realise that we can ask BASIC to refer to any element of the array simply by changing the value of the variable month.

Taking many of the features of the previous sections we can now rewrite the earlier program very neatly by using month as an index.

```
10 DIM pay(3), bills(3)
20 LET month = 1
30 REPeat get_figures
40   READ pay(month), bills(month)
50   PRINT "surplus="; pay(month) - bills(month)
60   LET month = month + 1
70   IF month > 3 THEN EXIT get_figures
80 END REPeat get_figures
90 DATA 987.32, 323.32, 942.27, 876.11, 966.39, 892.15
```

Curiously, this program is still longer than the previous one, yet this is a small price for its flexibility. To change the previous version of the program to handle 12 months requires a lot of typing, the second requires only two lines to be changed:

```
10 DIM pay(12), bills(12)
70 IF month > 12 THEN EXIT get_figures
```

in addition to adding more **DATA** statements.

FOR loops

From your understanding of BASIC you will realise that lines 20, 30, 60, 70, 80 are all concerned with increasing the value of month a certain number of times. In general, the outline for this is:

```
LET variable = initial_value
  REPeat loop
     IF variable > highest_value THEN EXIT loop
     various statements
     LET variable = variable + some_value
  END REPeat loop
```

This is rather long-winded considering that BASIC programs frequently use this construct, so TONTO BASIC allows you to write it more simply:

```
FOR variable=initial_value TO highest_value STEP some_value
    various statements
END FOR variable
```

For example:

```
FOR month = 1 TO 3 STEP 1
    READ pay(month)
    PRINT "tax:"; pay(month) * 32/100
END FOR month
```

To simplify things even further BASIC allows you to miss out the STEP *some_value* portion and will assume it is 1.

Because month is used to control the exit condition of the loop it is often referred to as the **control variable** or **loop variable**.

Try this program (remember to type **NEW** first):

```
10 CLS
20 FOR i = 1 TO 12
30   PRINT i,
40 END FOR i
```

and you will have just taught your TONTO to count!

String arrays

As an array is just a group of related variables you may make TONTO BASIC store strings in the same way as numbers. Just as a string variable ends in a $, so must a **string array variable**, so:

```
DIM month$ (3,10)
```

As mentioned earlier, this declaration of month$ has an extra number. The last number or dimension of a string array tells TONTO BASIC the maximum number of characters in each element of the string array, so that it doesn't use more store than it needs to.

There are two points to note here:

1 Although an array can hold strings you cannot refer to an element using a string index. Pay ("bacon") doesn't make much sense to anyone yet, and while you might make something of *pay ("Peb")*, TONTO BASIC cannot.

2 Do not confuse the extra number on the **DIM**ension of string arrays:

```
DIM month$ (3,10)
```

does not give you a table of strings three rows by ten columns.

If you wish to think of it another way, it will give you a 3 by 10 table of single characters.

SELF TEST ON SECTION 6

You can score a maximum of 10 points from the following test. Check your score with the answers on the next page.

1 What difficulty can arise when the data needed for a program becomes numerous and you try to handle it without arrays?

2 How do you tell TONTO BASIC what size your array will be? (2 points)

3 What is the word for the number or variable that distinguishes a particular element of an array?

4 Can you think of two ideas in ordinary life which correspond to the concept of an array in programming? (2 points)

5 A **REPeat** loop needs a name so that you can **EXIT** to its **END** properly. A **FOR** loop also has a name but what other function does a **FOR** loop's name have?

6 What are the two phrases which are used to describe the variable which is also the name of a **FOR** loop? (2 points)

7 The values of a loop variable change automatically as a **FOR** loop is executed. Name one possible important use of these values.

ANSWERS TO SELF TEST ON SECTION 6

1 Apart from the problems of choosing many different names for your variables, you cannot take advantage of loops to process your data

2 You must declare an array giving its size (dimension) using a **DIM** statement before you use it. (1 point)

If the array is a string array you must add another dimension to tell BASIC how long each string can be
(1 point)

3 The distinguishing number or variable is called an **index**, though some people may call it a **subscript**. Just remember how an index in a book lets you find the required page

4 Houses in a street share the same street name but each has its own number.

Beds in a hospital ward may share the name of the ward but each bed may be numbered.

Cells in a prison block may have a common block name but a different number.

Holes on a golf course (e.g. the fifth hole of Royal Birkdale is the fifth element in an array of 18)
(2 points)

5 A **FOR** loop's name is also the name of the variable which controls the loop

6 The two phrases for this variable are loop variable or control variable (2 points)

7 The values of a loop variable may be used as subscripts for array variable names. Thus, as the loop proceeds, each array element is visited once

CHECK YOUR SCORE

This test is more searching than the previous ones.

9 or 10 is excellent. Carry on reading.

7 or 8 is very good but think a bit more about some of the ideas. Look at programs to see how they work.

5 or 6 is good but re-read some parts of section 6.

3 to 4 is fair but re-read some parts of section 6 and do the test again.

Under 3, you should re-read section 6 carefully and do the test again.

EXERCISES ON SECTION 6

1 Use a **FOR** loop to place one of four numbers 1,2,3,4 randomly in five array variables:

card(1), card(2), card(3), card(4), card(5)

It does not matter if some of the four numbers are repeated. Use a second **FOR** loop to output the values of the five card variables

2 Imagine that the four numbers 1,2,3,4 represent 'Hearts', 'Clubs', 'Diamonds', 'Spades'. What extra program lines would need to be inserted to get output in the form of these words instead of numbers?

3 Use a **FOR** loop to place five random numbers in the range 1 to 13 in an array of five variables:

card(1), card(2), card(3), card(4) and card (5)

Use a second **FOR** loop to output the values of the five card variables

4 Imagine that the random numbers generated in exercise 3 represent cards. Write down the extra statements that would cause the following output:

Number	Output
1	the word 'Ace'
2 to 10	the actual number
11	the word 'Jack'
12	the word 'Queen'
13	the word 'King'

7 Simple procedures

MODULARITY

When you are writing computer programs to solve complex problems, it can be difficult to keep track of things. A methodical problem-solver therefore divides a large or complex job into smaller sections or tasks, and then divides these tasks again into smaller tasks, and so on until each can be easily tackled.

This is similar to the arrangement of complex human affairs. Successful government depends on a delegation of responsibility. The Prime Minister divides the work amongst ministers, who divide it further through the Civil Service until tasks can be done by individuals without further division. There are complicating features such as common services and interplay between the same and different levels, but the hierarchical structure is the dominant one.

A good programmer also works in this way and an advanced language like TONTO BASIC, which allows properly named, well defined procedures to deal with individual tasks, is much more helpful than older versions which do not have such features.

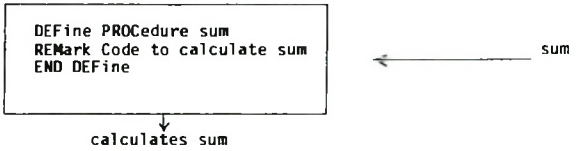
WHAT ARE PROCEDURES

You can use procedures to avoid duplication in programs and aid their readability. Use of procedures encourages **block structured programming**. The idea is that a separately named block of code should be written for a particular task. It doesn't matter where the block of code is in the program. If it is there somewhere, the use of its name:

- Activates the code
- Returns control to the point in the program immediately after that use

Such a block of code, composed of a sequence of statements which define a particular task, is called a **PROCedure**. A **PROC**edure is given a name and used when needed, as many times as necessary during the execution of a program.

If a procedure, sum, adds the numbers in an array, the scheme is as shown below:



In practice you can identify and name the separate tasks within a job before the definition code is written. The name is all you need to call the procedure so you can write the main outline of the program before all the tasks are defined.

Alternatively, you can write and test the tasks first. If the procedure works, you can then forget the details and just remember the name and what it does.

USE OF PROCEDURES

The following example could quite easily be written without procedures but it shows how they can be used in a reasonably simple context. Almost any task can be broken down in a similar fashion, which means that you never have to worry about more than, say, five to thirty lines at any one time. If you can write thirty-line programs well and handle procedures, you have the capability to write three-hundred-line programs.

PROCEDURES IN PROGRAMS

Example

You can produce ready made 'buzz-phrases' for politicians or others who wish to give an impression of technological fluency without actually knowing anything! Store the following words in three arrays and then produce ten random buzz-phrases.

adjec1$	adjec2$	noun$
Full	fifth-generation	systems
Systematic	knowledge-based	machines
Intelligent	compatible	computers
Controlled	cybernetic	feedback
Automated	user-friendly	transputers
Synchronised	parallel	micro-chips
Functional	learning	capability
Optional	adaptable	programming
Positive	modular	packages
Balanced	structured	databases
Integrated	logic-oriented	spreadsheets
Coordinated	file-oriented	word-processors
Sophisticated	standardised	objectives

Program Analysis and Design

To write a program to produce ten buzzword phrases, the stages of the program are:

1 Store the words in three string arrays

2 Choose three random numbers which will be the subscripts of the array variables

3 Print the phrase

4 Repeat 2 and 3 ten times

Variables

Identify three arrays of which the first two contain adjectives or words used as adjectives (describing words). The third array holds the nouns. There are 13 words in each section and the longest word has 16 characters including a hyphen.

Array	Purpose
adject1$(13,13)	first adjectives
adject2$(13,16)	second adjectives
noun$(13,15)	nouns

Procedures

Use three procedures to match the tasks identified.

store_data stores the three sets of thirteen words

get_random gets three random numbers in range 1 to 13

make_phrase prints a phrase

Main program

The program is very simple because the main work is done by the procedures.

Program design

```
1     Declare (DIM) the arrays

2     Store_data

3     FOR ten phrases
      get_random
      make_phrase
      END
```

Program

```
10  REMark ************
20  REMark * Buzzword *
30  REMark ************
40  DIM adjec1$(13,13), adjec2$(13,16),noun$(13,15)
50  store_data
60  FOR phrase = 1 TO 10
70    get_random
80    make_phrase
90  END FOR phrase
95  STOP
100 REMark ***************************
110 REMark * Procedure Definitions  *
120 REMark ***************************
130 DEFine PROCedure store_data
140    REMark *** procedure to store the buzzword data ***
150    FOR item = 1 TO 13
160      READ adjec1$(item), adjec2$(item),noun$(item)
180    END FOR item
190 END DEFine
200 DEFine PROCedure get_random
210   REMark *** procedure to select the phrase ***
220   LET ad1 = RND(1 TO 13)
230   LET ad2 = RND(1 TO 13)
240   LET n = RND(1 TO 13)
```

```
250 END DEFine
260 DEFine PROCedure make_phrase
270   REMark *** procedure to print out the phrase ***
280   PRINT ! adjec1$(ad1) ! adjec2$(ad2) ! noun$(n)
290 END DEFine
300 REMark ****************
310 REMark * Program Data *
320 REMark ****************
330 DATA "Full", "fifth-generation","systems"
340 DATA "Systematic", "knowledge-based", "machines"
350 DATA "Intelligent", "compatible", "computers"
360 DATA "Controlled", "cybernetic", "feedback"
370 DATA "Automated", "user-friendly", "transputers"
380 DATA "Synchronised", "parallel", "micro-chips"
390 DATA "Functional", "learning", "capability"
400 DATA "Optional", "adaptable", "programming"
410 DATA "Positive", "modular", "packages"
420 DATA "Balanced", "structured", "databases"
430 DATA "Integrated", "logic-oriented", "spreadsheets"
440 DATA "Coordinated", "file-oriented", "word-processors"
450 DATA "Designed", "standardised", "objectives"
```

Sample output

```
Automated fifth-generation capability
Functional learning packages
Full parallel objectives
Positive user-friendly spreadsheets
Intelligent file-oriented capability
Synchronised cybernetic transputers
Functional logic-oriented micro-chips
Positive parallel feedback
Balanced learning databases
Controlled cybernetic objectives
```

PASSING INFORMATION TO PROCEDURES

Suppose you wish to generate three random numbers in a given range.

To define a procedure rand you need five items of information:

- names of three variables to receive the random numbers
- the lower and upper limits of the numbers to be generated

The procedure definition would be as follows

```
200 DEF PROC rand (r1, r2, r3, low, high)
210 REMark * procedure to select the number *
220   LET r1 = RND (low TO high)
230   LET r2 = RND (low TO high)
240   LET r3 = RND (low TO high)
250 END DEF rand
```

r1, *r2*, *r3*, low and high are known as **formal parameters**. As yet they have no value or type.

When this procedure is called these formal parameters take on the values supplied by the calling statement. This action is known as **parameter substitution**. The values, supplied by the calling statement, are called the **actual parameters**.

For example, you might add the following program lines to print three random numbers.

```
10 CLS
20 rand x, y, z, 15, 32
30 PRINT x, y, z
```

The substitution (or passing of parameters) can be thought of as implicit LET statements thus:

```
201 LET low = 15 : high = 32
249 LET x=r1 : y=r2 : z=r3
```

The advantages of procedures are:

1 You can use the same code more than once in the same program or in others

2 You can break down a task into sub-tasks and write procedures for each sub-task. This helps the analysis and design

3 Procedures can be tested separately. This helps the testing and debugging

4 Meaningful procedure names and clearly defined beginnings and ends help to make a program readable

When you get used to properly named procedures with good parameter facilities, you should find that your problem-solving and programming powers are greatly enhanced.

SELF TEST ON SECTION 7

You can score a maximum of 14 points from the following test. Check your score with the answers on the following page.

1 How do we normally tackle the problem of great size and complexity in human affairs?

2 How can this principle be applied in programming?

3 What are the two most obvious features of a simple procedure definition? (2 points)

4 What are the two main effects of using a procedure name to 'call' the procedure? (2 points)

5 What is the advantage of using procedure names in a main program before the procedure definitions are written?

6 What is the advantage of writing a procedure definition before using its name in a main program?

7 How can the use of procedures help a 'thirty-line-programmer' to write much bigger programs?

8 Some programs use more memory in defining procedures, but in what circumstances do procedures save memory space?

9 What two methods can be used to pass information between a procedure and the main program? (2 points)

10 What are actual parameters?

11 What are formal parameters?

ANSWERS TO SELF TEST ON SECTION 7

1 We normally break down large or complex jobs into smaller tasks until they are small enough to be completed

2 This principle can be applied in programming by breaking the total job down and writing a procedure for each task

3 A simple procedure is:

a separate block of code
properly named (2 points)

4 A procedure call ensures that:

the procedure is activated
control returns to just after the calling point (2 points)

5 Procedure names can be used in a main program before the procedures have been written. This enables you to think about the whole job and get an overview without worrying about the detail

6 If you write a procedure definition before using its name you can test it and then when it works properly forget the details. You need only remember its name and what it does, not how it works

7 A programmer who can write up to thirty line programs can break down a complex task into procedures in such a way that none is more than thirty lines and most are much less. In this way he need only worry about one bit of the job at a time

8 The use of a procedure saves memory space if it is necessary to call it more than once from different parts of a program. The definition of a procedure only occurs once but it can be called as often as necessary

9 a) A procedure and program can use the same variables (pigeon-holes). These variables can then be updated by using either **LET**, **READ** and **INPUT** statements

b) A procedure and program may pass information to each other by using parameter substitution. When the procedure is activated BASIC copies the values from the main program into the procedures formal parameter variables. On completion of the procedure these, possibly updated, values may be copied back to the main programs variables
(2 points)

10 An actual parameter is the actual value passed from a procedure call in a main program to a procedure

11 A formal parameter is a variable in a procedure definition which represents a value or variable passed to the procedure by the main program

---

CHECK YOUR SCORE

This is a searching test. You may need more experience of using procedures before the ideas can be fully appreciated. But they are very powerful and, when understood, extremely helpful ideas. They are worth whatever effort is necessary.

12 to 14 excellent. Read on with confidence

10 or 11 very good. Just check again on certain points

8 or 9 good but re-read some parts of section 7

6 or 7 fair but re-read some parts of section 7. Work carefully through the programs writing down all changes in variable values. Then do the test again

Under 6 read section 7 again. Take it slowly, working through all the programs. These ideas may not be easy but they are worth the effort. When you are ready, take the test again

---

EXERCISES ON SECTION 7

1 Write a procedure which outputs one of the four suits: 'Hearts', 'Clubs', 'Diamonds', or 'Spades'. Call the procedure five times to get five random suits

2 Write a program of exercise 1 using a number in the range 1 to 4 as a parameter to determine the output word. If you have already done this, then try writing the program without parameters

3 Write a procedure which outputs the value of a card that is a number in the range 2 to 10 or one of the words 'Ace', 'Jack', 'Queen', 'King'

4 Write a program which calls this procedure five times so that five random values are output

5 Write the program of exercise 3 again using a number in the range 1 to 13 as a parameter to be passed to the procedure. If this was the method you used first time, try writing the program without parameters

6 Write the most elegant program you can, using procedures, to output four hands of five cards each. Do not worry about duplicate cards. You can take elegance to mean an appropriate mixture of readability, shortness, adaptability and performance. Different people and/or different circumstances will place different importance on these qualities, which sometimes work against each other

8 From BASIC to TONTO BASIC

INTRODUCTION

If you are familiar with one of the earlier version of BASIC you may be able to omit the first seven sections of Part B and use this section as a bridge between what you know already and the remaining sections. If you do this and still find areas of difficulty,it may be helpful to backtrack a little into some of the earlier sections.

If you have worked through the earlier sections this one should be easy reading. You may find that, as well as introducing some new ideas, it gives an interesting slant on the way BASIC is developing. Apart from its program structuring facilities, TONTO BASIC advances the frontiers of good screen presentation, editing, and operating facilities. In short, it is a combination of user-friendliness and computing power which has not existed before.

So when you make the transition to TONTO BASIC you are moving not only to a more powerful, more helpful language, you are also moving into a remarkably advanced computing environment.

The main features of TONTO BASIC and the features which distinguish it from other BASICs are described below.

ALPHABETIC COMPARISONS

The usual simple arithmetic comparisons are possible. You can write:

```
LET pet1$ = "CAT"
LET pet2$ = "DOG"
IF pet1$ < pet2$ THEN PRINT "meow"
```

The output will be **meow** because in this context the symbol < means

is earlier than (i.e. nearer to A in the alphabet)

TONTO BASIC allows sensible comparisons to be made. For example you would probably expect:

'cat' to come before 'DOG'

and

'ABC2' to come before 'ABC10'

In TONTO BASIC if the string cannot be converted, an error is reported.

LOGICAL VARIABLES AND SIMPLE PROCEDURES

There is one other type of variable in TONTO BASIC or rather the TONTO system makes it seem so. Consider the statement:

```
IF Windy THEN fly_kite
```

In other BASICs you might write:

```
IF w=1 THEN GOSUB 300
```

In this case w=1 is a condition or logical expression which is either true or false. If it is true, a subroutine starting at line 300 is executed. This subroutine may deal with kite flying but you cannot tell from the above line. A careful programmer would write:

```
IF w=1 THEN GOSUB 300 : REM fly_kite
```

to make it more readable. But the TONTO BASIC statement is readable as it stands. The identifier windy is interpreted as true or false though it is actually a floating point variable. A value of 1 or any non-zero value is taken as true. Zero is taken as false. Thus the single word, windy, has the same effect as a condition or logical expression.

The other word, *fly_kite*, is a procedure. It does a good job similar to, but rather better than, GOSUB 300.

The following program conveys the idea of logical variables and the simplest type of named procedure.

```
INPUT windy
IF windy THEN fly_kite
IF NOT windy THEN tidy_shed
DEFine PROCedure fly_kite
  PRINT "See it in the air."
END DEFine
DEFine PROCedure tidy_shed
  PRINT "Sort out rubbish."
END DEFine
```

INPUT	OUTPUT
0	Sort out rubbish.
1	See it in the air
2	See it in the air
-2	See it in the air

You can see that only zero is taken as meaning false. You would not normally write procedures with only one action statement, but the program illustrates the idea and syntax in a very simple context. More is said about procedures later in this section.

LET STATEMENTS

In TONTO BASIC the LET keyword is optional but we use it in this manual so that there is less chance of confusion caused by the two possible uses of =. The meanings of = in:

```
LET count = 3  (let count become 3)
```

and in

```
IF count = 3 THEN EXIT  (if count is equal to 3)
```

are different and the LET helps to emphasise this. However, if there are two, or a few, LET statements doing some simple job such as setting initial values, an exception may be made.

For example:

```
20 LET first = 0
30 LET second = 1
40 LET third = 2
```

may be re-written as:

```
20 LET first = 0 : second = 1 : third = 2
```

without loss of clarity or style. It is also consistent with the general concept of allowing short forms of other constructions where they are used in simple ways.

The colon (:) is a valid statement separator and may be used with other statements besides **LET**.

THE BASIC SCREEN

This section describes the pixel-oriented features. A pixel is the smallest area of colour which can be displayed. The TONTO has two screen modes which are described below:

Modes and Pixels

```
Low resolution
8 Colour Mode
256 pixel across, 256 down
```

```
High resolution
4 Colour Mode
512 pixel across, 256 down
```

BASIC operates only in high resolution mode and uses an area of 480 x 240 pixels within the TONTO screen.

Pixels are addressed by the range of numbers:

0 - 479 across
and **0 - 239 down**

COLOUR TONES

The colour tones available are:

Colour code	Display tone
0 1	black
2 3	dark grey
4 5	light grey
6 7	white

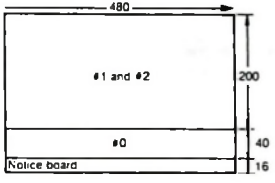
Some of the screen presentation keywords are as follows:

INK colour — foreground colour

PAPER colour — background colour

TONTO SCREEN ORGANISATION

When you first enter BASIC, and after **NEW**, the application screen display is split into three areas called windows:



The windows are identified by #0,#1 and #2 so that you can relate various effects to particular windows. For example:

```
CLS
```

clears window 1 (the default display window) so if you want the bottom area cleared you must type:

```
CLS #0
```

If you want a different **PAPER** (background colour) type for light grey

```
PAPER 4 : CLS
```

or

```
PAPER #2,4 : CLS #2
```

to clear window 2 to the background colour light grey.

The numbers #0, #1, #2 are called channel numbers. In this particular case they enable you to direct certain effects to the window of your choice. You will discover later that channel numbers have many other uses, but for the moment note that all of the following statements may have a channel number. The third column shows the default channel - the one chosen by the system if you do not specify one.

Note that windows 1 and 2 completely overlap. The system overlaps these windows so that more character positions per line are available for program listings.

Keyword	Effect	Default Channel
AT	Character position	#1
CLS	Clears screen	#1
CSIZE	Character size	#1
INK	Foreground colour	#1
PAPER	Background colour	#1
UNDER	Underlines	#1
WINDOW	Changes existing window	#1
LIST	Lists program	#2
PRINT	Prints characters	#1
INPUT	Takes keyboard input	#1
INKEY$	Takes keyboard character	#1

Statements or direct commands appear in window 0

For more information about these keywords see Keywords (part D).

INPUT AND OUTPUT

TONTO BASIC has the usual **LET**, **INPUT**, **READ** and **DATA** statements for input. The **PRINT** statement handles most text output in the usual way with the separators:

, tabulates output

; just separates - no formatting effect

\ forces new line

! intelligent space. Normally provides a space but not at the start of line. If an item will not fit at the end of a line it performs a new line operation

LOOPS

You may be familiar with two types of repetitive loop in BASIC exemplified as follows:

(a) Simulate 6 throws of an ordinary six-sided die.

```
10 FOR throw = 1 TO 6
20   PRINT RND(1 TO 6)
30 NEXT throw
```

(b) Simulate throws of a die until a six appears.

```
10 die = RND(1 TO 6)
20 PRINT die
30 IF die <> 6 then GOTO 10
```

Although these programs both work in TONTO BASIC we recommend using the following equivalent programs instead. They do exactly the same jobs as their BASIC counterparts. Although program (b) is a little more complex there are good reasons for preferring it.

(a)
```
10 FOR throw = 1 TO 6
20   PRINT RND(1 TO 6)
30 END FOR throw
```

(b)
```
10 REPeat throws
20   die = RND(1 TO 6)
30   PRINT die
40   If die = 6 THEN EXIT throws
50 END REPeat throws
```

REPeat loops

It is logical to provide a structure for a loop which terminates on a condition (**REPeat** loops) as well as those which are controlled by a count.

The fundamental **REPeat** structure is:

REPeat identifier
  statements
END REPeat identifer

The EXIT statement

The **EXIT** statement can be placed anywhere in the structure, but it must be followed by an identifier to tell TONTO BASIC which loop to exit; for example:

EXIT throws

transfers control to the statement after

END REPeat throws.

This may seem like using a sledgehammer to crack the nut of the simple program illustrated. However, the **REPEAT** structure is very powerful. If you know other languages you may see that it does the jobs of both **REPeat** and **WHILE** structures and also copes with other, more awkward, situations.

The TONTO BASIC **REPeat** loop is named so that a correct clear exit is made. The **FOR** loop, like all TONTO BASIC structures, ends with **END**, and its name is given for reasons which will become clear later.

You will also see later how these loop structures can be used in simple or complex situations to match exactly what you need to do. We mention only three more features of loops at this stage. They will be familiar if you are an experienced user of BASIC.

The STEP keyword

The increment of the control variable of a **FOR** loop is normally 1 but you can make it other values by using the STEP keyword, as the examples show.

```
(a)  10 FOR even = 2 TO 10 STEP 2
     20   PRINT ! even !
     30 END FOR even

     output is 2 4 6 8 10

(b)  10 FOR backwards = 9 TO 1 STEP - 1
     20   PRINT ! backwards !
     30 END FOR backwards

     output is 9 8 7 6 5 4 3 2 1
```

Nested loops

The second feature is that loops can be nested. You may be familiar with nested **FOR** loops. For example the following program outputs four rows of ten crosses.

```
10 REMark Crosses
20 FOR row = 1 TO 4
30   PRINT 'Row number'! row
40   FOR cross = 1 TO 10
50     PRINT "X"!
60   END FOR cross
70   PRINT
80   PRINT 'End of row number'! row
90 END FOR row
```

output is

```
Row number 1
X X X X X X X X X X
End of row number 1
Row number 2
X X X X X X X X X X
End of row number 2
Row number 3
X X X X X X X X X X
End of row number 3
Row number 4
X X X X X X X X X X
End of row number 4
```

A big advantage of TONTO BASIC is that it has structures for all purposes, not just FOR loops, and they can all be nested one inside the other, reflecting the needs of a task. We can put a REPeat loop in a FOR loop. The program below produces scores of two dice in each row instead of crosses, until a seven occurs.

```
 10 REMark Dice rows
 20 FOR row = 1 to 4
 30   PRINT 'row number'! row
 40   REPeat throws
 50     LET die1 = RND(1 TO 6)
 60     LET die2 = RND(1 TO 6)
 70     LET score = die1 + die2
 80     PRINT score!
 90     IF score = 7 THEN EXIT throws
100   END REPeat throws
110   PRINT 'End of row number' ! row
120 END FOR row
```

sample output:

```
Row number 1
8 11 6 3 7
End of row number 1
Row number 2
4 6 2 9 4 5 12 7
End of row number 2
Row number 3
7
End of row number 3
Row number 4
6 2 4 9 9 7
End of row number 4
```

Repetition

The third feature of loops in TONTO BASIC allows more flexibility in providing the range of values in a **FOR** loop. The following program illustrates this by printing all the non-prime numbers from 1 to 20.

```
10 REMark Divisible numbers
20 FOR num = 4, 6, 8 TO 10, 12, 14 TO 16, 18, 20
30   PRINT ! num!
40 END FOR num
```

More is said about handling repetition in a later section but the features described above handle all but a few uncommon or advanced situations.

DECISION MAKING

You will have noticed the simple type of decision:

```
IF die = 6 THEN EXIT throws
```

The IF...THEN statement

The **IF...THEN** statement is available in most BASICs but TONTO BASIC offers extensions of this structure, plus a completely new one for handling situations with more than two alternative courses of action.

However, you may find the following long forms of **IF...THEN** useful. They should explain themselves.

Program 1 IF...END IF

```
10 REMark Long form IF...END IF
20 LET sunny = RND(0 TO 1)
30 IF sunny THEN
40   PRINT "Wear sunglasses"
50   PRINT "Go for walk"
60 END IF
```

Program 2 IF...ELSE...END IF

```
10 REMark Long form IF...ELSE...END IF
20 LET sunny = RND(0 TO 1)
30 IF sunny THEN
40   PRINT "Wear sunglasses"
50   PRINT "Go for walk"
60 ELSE
70   PRINT "Wear coat"
80   PRINT "Go to cinema"
90 END IF
```

The separator, **THEN**, is optional in long forms or it can be replaced by a colon in short forms. The long decision structures have the same status as loops. You can nest them or put other structures into them. When a single variable appears where you expect a condition, the value zero is taken as false and other values as true.

SUBROUTINES AND PROCEDURES

Most BASICs have a **GOSUB** statement which may be used to activate particular blocks of code called subroutines. The **GOSUB** statement is unsatisfactory in a number of ways, so TONTO BASIC offers in its place properly named procedures with some very useful features.

Consider the following programs both of which draw a light grey *square* of side length 60 pixel screen units, at a position 180 across 100 down, on a dark grey background.

Program 1 Using GOSUB

```
 10 LET colour = 4 : background = 2
 20 LET across = 180
 30 LET down = 100
 40 LET side = 60
 50 GOSUB 80
 60 PRINT 'END'
 65 STOP
 70 REMark
 80 PAPER background : CLS
 90 OPEN#15, SCR_
100 WINDOW#15, side, side, across, down
110 PAPER#15, colour : CLS#15
120 CLOSE#15
130 RETURN
```

Program 2 Using a procedure with parameters

```
 10 square 4, 60, 180, 100, 2
 20 PRINT 'END'
 30 DEFine PROCedure square
                    (colour,side,across,down,background)
 40   PAPER background : CLS
 50   OPEN#15, SCR_
 60   WINDOW#15, side, side, across, down
 70   PAPER#15, colour : CLS
 80   CLOSE#15
 90 END DEFine
```

In Program 1 the values of colour, across, down and side are fixed by **LET** statements before the **GOSUB** statement activates lines 80 and 90. Control is then sent back by the **RETURN** statement.

In Program 2 the values are given in the first line as parameters in the procedure call, square, which activates the procedure and at the same time provides the values it needs.

In its simplest form, a procedure has no parameters. It merely separates a particular piece of code, though even in this simpler use the procedure has an advantage over **GOSUB** because it is properly named and properly isolated into a self-contained unit.

The power and simplifying effects of procedures are more obvious as programs get larger. What procedures do as programs get larger is not so much make programming easier, as prevent it from getting harder with increasing program size. The above example just illustrates the way they work in a simple context.

VOCABULARY AND SYNTAX OF TONTO BASIC

The following examples indicate the range of vocabulary and syntax of TONTO BASIC which has been covered in this and earlier sections, and forms a foundation on which the following parts of this manual build.

Example 1

The letters of a palindrome are given as single items in **DATA** statements. The terminating item is an asterisk and you assume no knowledge of the number of letters in the palindrome. **READ** the letters into an array and print them backwards. Some palindromes such as 'MADAM I'M ADAM' only work if spaces and punctuation are ignored. The one used here works properly.

Program

```
 10 REMark Palindromes
 20 DIM text$(30)
 30 LET count = 31
 40 REPeat get_letters
 50   READ character$
 60   IF character$ = '*' THEN EXIT get_letters
 70   LET count = count-1
 80   LET text$(count) = character$
 90 END REPeat get_letters
100 PRINT text$(count TO 30)
110 DATA 'A','B','L','E',' ','W','A','S',' ','I',' ','E','R'
120 DATA 'E',' ','I',' ','S','A','W',' ','E','L','B','A','*'
```

Output is:

```
ABLE WAS I ERE I SAW ELBA
```

Example 2

The following program accepts a positive number as input and converts it into an equivalent in Roman Numerals. It does not generate the most elegant form; that is it produces IIII rather than IV, VIIII instead of IX etc.

```
 10 REMark Roman numbers
 20 INPUT number
 30 FOR type = 1 TO 7
 40   READ letter$, value
 50   REPeat output
 60     IF number < value : EXIT output
 70     PRINT letter$;
 80     LET number = number - value
 90   END REPeat output
100 END FOR type
110 DATA 'M',1000,'D',500,'C',100,'L',50,'X',10,'V',5,'I',1
120 RESTORE
```

Sample output

Input	Output
3289	MMMCCLXXXVIIII
100	C
4	IIII
1542	MDXXXXII
17	XVII

You should study the above examples carefully, using dry runs if necessary, until you are sure that you understand them.

TONTO FEATURES AND FACILITIES

In TONTO BASIC full structuring features are provided, so that program elements either follow in sequence or fit into one another neatly. All structures must be identified to the system and named. There are many unifying and simplifying features and many extra facilities.

Most of these are explained and illustrated in the remaining sections of the Beginner's Guide and are easier to read here than in the Concept and Keyword Reference Guide (Parts C & D) However, these sections do not give every technical detail or exhaust every topic which they treat. There may be, therefore, a few occasions when you need to consult the reference parts. On the other hand some major advances are discussed in the following sections. Few readers will need to use all of them and you may find it helpful to omit certain parts, at least on first reading.

SELF TEST ON SECTION 8

1 Write a program which will accept five words as data and then print them in reverse order

2 Write a program which will accept five words as data and print the one which is first alphabetically

3 Modify the program of question 2 to print the word which is last alphabetically

4 Write a program which will accept five words as data, then search for a stated word and replace it, if it is found, with another stated word

5 Write a program which completes the following line in a variety of random ways

"The light ..... on the ....."

Using words chosen from

dances	molten	waves
gleams	soft	steel
leaps	arcing	water

ANSWERS TO SELF TEST ON SECTION 8

1
```
10 REMark Reverse Words
20 DIM word$(5,7)
30 FOR Num = 1 to 5 : READ word$(num)
40 FOR Num = 5 to 1 STEP -1 : PRINT word$(num)!
50 DATA "blue", "red", "magenta", "green", "cyan"
```

2
```
10 REMark Print First
20 LET first$ = "zzz"
30 FOR num = 1 to 5
40   READ word$
50   IF word$ < first$ THEN LET first$ = word$
60 END FOR num
65 PRINT first$
70 DATA "one", "two", "three", "four", "five"
```

3
```
10 REMark Print Last
20 LET last$ =  " "
30 FOR num = 1 to 5
40   READ word$
50   IF word$ > last$ THEN LET last$ = words$
60 END FOR num
65 PRINT last$
70 DATA "one", "two", "three", "four", "five"
```

4
```
 10 REMark word replacement
 20 DIM word$(5,16)
 30 CLS
 40 FOR num=1 TO 5
 50   READ word$(num)
 60 END FOR num
 70 REPeat replaceword
 80   PRINT "The words are:"
 90   FOR num=1 TO 5:PRINT !word$(num);
100   PRINT
110   INPUT "Which word do you wish to replace?" !target$
120   IF LEN(target$)=0 THEN EXIT replaceword
130   FOR num=1 TO 5
140     IF word$(num)=target$ THEN
150      REPeat getword
160       INPUT "Enter the word you wish to replace"!
                                        (word$(num);":"!replace$
170       IF LEN(replace$) < 17 THEN
180           word$(num)=replace$
190           EXIT num
200       END IF
210      END REPeat getword
220     END IF
230     NEXT num
240     PRINT target$!"is not in my vocabulary"
250     END FOR num
260 END REPeat replaceword
```

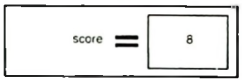
```
270 DATA "The","quick","sly","brown","fox"

5  100 REMark The light...on the...
   110 DIM word$ (3,3,6)
   120 RESTORE 230
   130 FOR type = 1 TO 3
   140   FOR word = 1 TO 3
   150     READ word$ (type, word)
   160   END FOR word
   170 END FOR type
   180 CLS
   190 FOR sentence = 1 TO 10
   200   PRINT "The light"! word$(1,RND(1 TO 3)! "on the";
   210   PRINT word$ (2,RND(1 TO 3))!word$(3,RND(1 TO 3));"."
   220 END FOR sentence
   230 DATA "dances", "gleams", "leaps"
   240 DATA "molten", "soft", "arcing"
   250 DATA "waves", "steel", "water"
```

9 Data types, variables and identifiers

You now know that a program (a sequence of statements) usually gets some data to work on (input) and produces some kind of results (output). You also understand that there are internal arrangements for storing this data. In order to avoid unnecessary technical explanations, we have suggested that you imagine data storage areas as pigeon-holes and that you choose meaningful names for these pigeon-holes. For example, if it is necessary to store a number which represents the score from two simulated dice-throws, you imagine a pigeon-hole named score which might contain a number such as 8.

Internally the pigeon-holes are numbered and the system maintains a dictionary which connects particular names with particular numbered pigeon-holes. We say that the name, score, points to its particular pigeon-hole (by means of the internal dictionary).



The whole arrangement is called a **variable**.

IDENTIFIERS AND VARIABLES

What you see is the word score. The word score is a variable name or **identifier**. It is what we see and it defines the concept we need, in this case the result, 8, of throwing a pair of dice. Because the identifier is what we see it becomes the thing we talk or write or think about. We write about score and its value at any particular moment. Rules for identifiers and variables are listed below.

1 An identifier must begin with a letter and is a sequence of:

   upper or lower case letters
   digits or underscore

2 An identifier may be up to 255 characters in length

3 An identifier cannot be the same as a BASIC keyword or a TONTO BASIC reserved word (see *Appendix 2*)

4 An integer variable name is an identifier with % as its last character

5 A string variable name is an identifier with $ as its last character

6 No other identifiers must use the symbols % or $

7 An identifier should be chosen so that it means something to the reader. For BASIC it does not have any meaning other than that it identifies variables

DATA TYPES

There are four simple data types called **floating point, integer, string** and **logical** which are explained below. We talk about data types, rather than variable types, because data can occur on its own (for example 3.4 or 'Blue hat') or as the value of a variable. Different types of variables hold corresponding types of data.

Floating point variables

Examples of the use of floating point variables are:

```
10 LET days = 24
20 LET sales = 3649.84
30 LET sales_per_day = sales/days
40 PRINT sales_per_day
```

The value of a floating point variable may be anything in the range:

-10^{615} to + 10^{615} with 8 significant figures

Suppose in the above program sales were, exceptionally, only 3p. Change line 10 to:

```
20 LET sales = 0.03
```

The system, when relisting the line, changes this to:

```
20 LET sales = 3E-2
```

To interpret this, start with 3 or 3.0 and move the decimal point -2 places, that is, two places left. This shows that:

3E-2 is the same as 0.03

After running the program, the average daily sales are shown to be:

1.25E-3 which is the same as 0.00125

Numbers with an E are said to be in exponent form:

(mantissa) E (exponent) = (mantissa) x 10 to the power (exponent)

Integer variables

Integer variables can have only whole number values in the range -32768 to 32767. The following are examples of valid integer variable names, which must end with %.

```
LET count% = 10
LET six_tally% = RND(10)
LET number_3% = 3
```

The only disadvantage of integer variables, when whole numbers are required, is the slightly misleading % symbol on the end of the identifier. It has nothing to do with the concept of percentage. It is just a convenient symbol tagged on to show that the variable is an integer.

Numeric functions

Using a function is a bit like making an omelette. You put in an egg which is processed according to certain rules (the recipe) and get out an omelette. For example, the function **INT** takes any number as input and outputs the greatest whole number that is less than or equal to the given number. For positive values this is equivalent to the integer part of the number. Anything which is input to a function is called a parameter or argument. If you write:

```
PRINT INT(5.6)
```

5 is output. We say that 5.6 is the parameter and the function returns the value 5. A function may have more than one parameter. You have already met:

```
RND(1 TO 6)
```

which is a function with two parameters. But functions always return exactly one value. This must be so, because you can put functions into expressions. For example:

```
PRINT 2 * INT(5.6)
```

produces the output 10. It is an important property of functions that you can use them in expressions. It therefore follows that they must return a single value which is then used in the expression. **INT** and **RND** are system functions; they come with the system, but later you will see how to write your own functions.

Trigonometrical functions are dealt with in a later section. Other common numeric functions are given in the list below.

Function	Effect	Example	Returned values
ABS	Absolute or unsigned value	ABS(7) ABS(-4.3)	7 4.3
INT	Integer part of a floating point number	INT(2.4) INT(0.4) INT(-2.7)	2 0 -3
SQRT	Square root	SQRT(2) SQRT(16) SQRT(2.6)	1.414214 4 1.612452

Numeric operations

TONTO BASIC allows the usual mathematical operations. You may notice that they are like functions with exactly two operands each. It is also conventional in these cases to put an operand on each side of the symbol. Sometimes the operation is denoted by a symbol such as + or *. Sometimes the operation is denoted by a keyword like **DIV** or **MOD** but there is no real difference. Numeric operations have an order of priority. For example, the result of:

```
PRINT 7 + 3*2
```

is **13** because the multiplication has a higher priority. However:

```
PRINT (7 + 3)*2
```

outputs **20**, because brackets override the usual priority. As you will see later, so many things can be done with TONTO BASIC expressions that a full statement about priority cannot be made at this stage (see the Concept Reference Guide if you want further information). The operations we now deal with have the following order of priority:

highest raising to a power
multiplication and division (including **DIV**, **MOD**)
lowest add and subtract

The symbols + and - are also used with only one operand which simply denotes positive or negative. Symbols used in this way have the highest priority of all and can only be overridden by the use of brackets.

Finally, if two symbols have equal priority the leftmost operation is performed first so that:

```
PRINT 7-2 + 5
```

causes the subtraction before the addition. This might be important if you should ever deal with very large or very small numbers.

The numeric operations you can use in TONTO BASIC are tabulated below:

Operation	Symbol	Examples	Results	Notes
Add	+	7+6.3	13.3	
Subtract	-	7-6.3	0.7	
Multiply	*	3*2.1 2.1*(-3)	6.3 -6.3	
Divide	/	7/2 -17/5	3.5 -3.4	Do not divide by zero
Raise to power	^	4^1.5	8	
Integer divide	DIV	-8 DIV 2 7 DIV 2 -7 DIV 2	-4 3 -4	Integers Only. Do not divide by zero
Modulus	MOD	13 MOD 5 21 MOD 7 -17 MOD 8	3 0 -7	Integers Only. Do not take modulus of zero

Numeric expressions

Strictly speaking, a numeric expression is an expression which evaluates to a number and there are more possibilities than we need to discuss here. TONTO BASIC allows you to do complex things if you want to, but it also allows you to do simple things in simple ways. In this section we concentrate on the straightforward uses of mathematical features.

Basically, numeric expressions in TONTO BASIC are the same as those of mathematics but you must put the whole expression in the form of a sequence. Thus:

$$\frac{5 + 3}{6 - 4}$$

becomes in TONTO BASIC

```
(5 + 3)/(6-4)
```

Example 1

In algebra there is an equation to find one root of a quadratic equation:

$$ax^2 + bx + c = 0$$

One solution in mathematical notation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If we start with the equation:

$$2x^2 - 3x + 1 = 0$$

The following program finds one solution.

```
10 READ a,b,c
20 PRINT 'Root is' ! (-b + SQRT(b ^ 2 - 4*a*c))/(2*a)
30 DATA 2,-3,1
```

Example 2

In problems which need to simulate the dealing of cards, you can make cards correspond to the numbers 1 to 52 as follows:

```
 1 to 13    Ace, two.......king of hearts
14 to 26    Ace, two.......king of clubs
27 to 39    Ace, two.......king of diamonds
40 to 52    Ace, two.......king of spades
```

A particular card, say 23, can be identified as follows:

```
 10 REM Card identification
 20 LET card = 23
 30 LET suit = (card-1) DIV 13
 40 LET value = card MOD 13
 50 IF value = 0 THEN LET value = 13
 60 IF value = 1 THEN PRINT "Ace of ";
 70 IF value >= 2 AND value <= 10 THEN PRINT value ! "of ";
 80 IF value = 11 THEN PRINT "Jack of ";
 90 IF value = 12 THEN PRINT "Queen of ";
100 IF value = 13 THEN PRINT "King of ";
110 IF suit = 0 THEN PRINT "hearts"
120 IF suit = 1 THEN PRINT "clubs"
130 IF suit = 2 THEN PRINT "diamonds"
140 IF suit = 3 THEN PRINT "spades"
```

There is a new idea in this program: it is in line 70. The meaning is clearly that the number is actually printed only if two logical statements are true. These are:

```
value is greater than (>=2) or equal to 2
AND value is less than (>=10) or equal to 10
```

Cards outside this range are either aces or court cards and must be treated differently.

Note also the use of ! in the **PRINT** statement to provide a space and ; to ensure that output continues on the same line.

There are two groups of mathematical functions which we have not discussed here. They are the trigonometric and logarithmic. Types of functions are also fully defined in the Concept Reference Guide.

Logical variables

Strictly speaking, TONTO BASIC does not allow logical variables, but it allows you to use other variables as logical ones. For example, you can run the following program:

```
10 REMark Logical Variable
20 LET hungry = 1
30 IF hungry THEN PRINT "Have a bun"
```

You expect a logical expression in line 30 but the numeric variable, hungry is there on its own. The system interprets the value, 1, of *hungry* as true and the output is:

```
Have a bun
```

If line 20 read:

```
Let hungry = 0
```

there would be no output. The system interprets zero as false and all other values as true. That is useful, but you can disguise the numeric quality of hungry by writing:

```
10 REMark Logical Variable
20 LET true = 1 : false = 0
30 LET hungry = true
40 IF hungry THEN PRINT "Have a bun"
```

String variables

There is much to be said about handling strings and string variables and this is left to a separate chapter.

EXERCISES ON SECTION 9

1 A rich oil dealer gambles by tossing a coin in the following way. If it comes down heads he get 1. If it comes down tails, he throws again but the possible reward is doubled. This is repeated so that the rewards are as shown.

THROW	1	2	3	4	5	6	7
REWARDS	1	2	4	8	16	32	64

By simulating the game, try to decide what would be a fair initial payment for each such game:

(a) if the player is limited to a maximum of seven throws per game

(b) if there is no maximum number of throws

2 Bill and Ben agree to gamble as follows. At a given signal each divides his money into two halves and passes one half to the other player. Each then divides his new total and passes half to the other. Show what happens as the game proceeds if Bill starts with 16p and Ben starts with 64p.

3 What happens if the game is changed so that each hands over an amount equal to half of what the other possesses?

4 Write a program which forms random three-letter words chosen from A, B, C, D and prints them until 'BAD' appears

5 Modify the last program so that it terminates when any real three letter word appears.

10 Logic

WHY LOGIC

From what you have read in previous chapters, you will probably agree that repetition, decision making and breaking tasks into sub-tasks are major concepts in problem analysis, program design and encoding programs. Two of these concepts, repetition and decision making, need logical expressions such as those in the following program lines:

```
IF score = 7 THEN EXIT throws
IF suit = 3 THEN PRINT "spades"
```

The first enables **EXIT** from a **REPeat** loop. The second is simply a decision to do something or not. A mathematical expression evaluates to one of millions of possible numeric values. Similarly, a string expression can evaluate to millions of possible strings of characters. You may find it strange that logical expressions, for which great importance is claimed, can evaluate to one of only two possible values: true or false.

In the case of:

```
score = 7
```

this is obvious. Either score equals 7 or it doesn't. The expression must be true or false, assuming that it's not meaningless. It may be that you do not know the value at some time, but that will be put right in due course.

LOGICAL OPERATORS

You have to be a bit more careful of expressions involving words such as **OR**, **AND**, and **NOT** but they are well worth investigating - indeed, they are essential to good programming. They will become even more important with the trend towards other kinds of languages based more on precise descriptions of what you require, rather than what the computer must do.

AND

The word **AND** in TONTO BASIC is like the word and in ordinary English. Consider the following program.

```
10 REMark AND
20 PRINT "Enter two values, 1 for TRUE or 0 for FALSE"
30 INPUT raining, hole_in_roof
40 IF raining AND hole_in_roof THEN PRINT "Get wet"
```

As in real life, you will get wet if it is raining and there is a hole in the roof. If one (or both) of the simple logical variables

raining
hole_in_roof

is false then the compound logical expression

raining ***AND*** *hole_in_roof*

is also false. It takes two true values to make the whole expression true. Only when the compound expression is true do you get wet. This can be seen from the rules for **AND** below:

raining	hole in roof	raining **AND** hole in roof	effect
FALSE	FALSE	FALSE	DRY
FALSE	TRUE	FALSE	DRY
TRUE	FALSE	FALSE	DRY
TRUE	TRUE	TRUE	WET

OR

In every day life the word or is used in two ways. We can illustrate the inclusive use of **OR** by thinking of a cricket captain looking for players. He might ask "Can you bat or bowl?" He would be pleased if a player could do just one thing well but would also be pleased if someone could do both. So it is in programming: a compound expression using **OR** is true if either or both of the simple statements or values of a variable are true. Try the following program.

```
10 REMark OR test
20 PRINT "Enter two values, 1 for TRUE or 0 for FALSE"
30 INPUT "Can you bat", batsman
40 INPUT "Can you bowl", bowler
50 IF batsman OR bowler THEN PRINT "In the team"
```

The rules for **OR** are listed below:

batsman	bowler	batsman OR bowler	effect
FALSE	FALSE	FALSE	not in team
FALSE	TRUE	TRUE	in the team
TRUE	FALSE	TRUE	in the team
TRUE	TRUE	TRUE	in the team

When the inclusive **OR** is used, a true value in either of the simple statements produces a true value in the compound expression. If Ian Botham, the England all-rounder, were to answer the questions both as a bowler and as a batsman, both simple statements would be true and so would the compound expression. He would be in the team.

If you write 0 for false and 1 for true you get all the possible combinations by counting in binary:

00 01 10 11

NOT and brackets The word **NOT** has the obvious meaning.

NOT true is the same as false
NOT false is the same as true

However you need to be careful. Suppose you hold a red triangle and say that it is:

NOT *red* **AND** *square*

In English this may be ambiguous. If you mean:

(**NOT** *red*) **AND** *square*

then for a red triangle the expression is false. If you mean:

NOT (*red* **AND** *square*)

for a red triangle the whole expression is true. There must be a rule in programming to make it clear what is meant. The rule is that **NOT** takes precedence over **AND** so the interpretation:

(NOT *red*) **AND** *square*

is the correct one. This is the same as:

NOT *red* **AND** *square*

To get the other interpretation you must use brackets. If you need to use a complex logical expression, it is best to use brackets and **NOT** if their usage naturally reflects what you want. But you can if you wish always remove brackets by using the following laws (attributed to Augustus De Morgan):

NOT (a **AND** b)	is the same as	**NOT** a **OR NOT** b
NOT (a **OR** b)	is the same as	**NOT** a **AND NOT** b

For example:

NOT (*tall* **AND** *fair*) is the same as
NOT *tall* **OR NOT** *fair*

NOT (*hungry* **OR** *thirsty*) is the same as
NOT *hungry* **AND NOT** *thirsty*

Test this by entering:

```
10 REMark NOT and brackets
20 PRINT "Enter two values, 1 for TRUE or 0 for FALSE"
30 INPUT "tall"; tall
40 INPUT "fair"; fair
50 IF NOT (tall AND fair) THEN PRINT "FIRST"
60 IF NOT tall OR NOT fair THEN PRINT "SECOND"
```

Whatever combination of numbers you give as input, the output is always either two words or none, never one. This suggests that the two compound logical expressions are equivalent.

XOR - Exclusive OR

Suppose a golf professional wanted an assistant who could either run the shop or give golf lessons. If an applicant turned up with both abilities, he might not get the job because the golf professional could fear that such an able assistant would try to take over. He would accept a good golfer who could not run the shop. He would also accept a poor golfer who could run the shop. This is an exclusive **OR** situation: either is acceptable but not both. The following program would test applicants:

```
10 REMark XOR test
20 PRINT "Type 1 for yes or 0 for no"
30 INPUT "Can you run a shop?", shop
40 INPUT "Can you teach golf?", golf
50 IF shop XOR golf THEN print "Suitable"
```

The only combinations of answers that cause the output "Suitable" are (0 and 1) or (1 and 0). The rules for XOR are given below:

Able to run shop	Able to teach	Shop XOR teach	effect
FALSE	FALSE	FALSE	no job
FALSE	TRUE	TRUE	gets the job
TRUE	FALSE	TRUE	gets the job
TRUE	TRUE	FALSE	no job

Priorities

The order of priority for the logical operators is (highest first):

NOT
AND
OR,XOR

For example the expression

rich **OR** *tall* **AND** *fair*

means the same as:

rich **OR** (*tall* **AND** *fair*)

The **AND** operation is performed first. To prove that the two logical expressions have identical effects run the following program:

```
10 PRINT "Enter three values, 1 for TRUE or 0 for FALSE"
20 REMark Priorities
30 INPUT rich,tall,fair
40 IF rich OR tall AND fair THEN PRINT "YES"
50 IF rich OR (tall AND fair) THEN PRINT "AYE"
```

Whatever combination of three zeros or ones you input at line 30 the output is either nothing or:

```
YES
AYE
```

You can make sure that you test all possibilities by entering data which forms eight three-digit binary numbers 000 to 111:

000 001 010 011 100 101 110 111

EXERCISES ON SECTION 10

1 Place ten numbers in a **DATA** statement. **READ** each number and if it is greater than 20 then print it

2 Test all the numbers from 1 to 100 and print only those which are perfect squares or divisible by 7

3 Toys are described as Safe (S), or Unsafe (U), Expensive (E) or Cheap (C), and suitable for young children (Y), teenagers (T) or all ages (A). A trio of letters encodes the qualities of each toy. Place five such trios in a **DATA** statement and then search it, printing only those which are safe and suitable for young children

4 Modify program 3 to print those which are expensive and not safe

5 Modify program 3 to print those which are safe, not expensive and suitable for anyone

6 Try to modify program 3 to accept a condition and print out those toys which satisfy this condition. Warning: this is not an easy exercise!

11 Handling text-strings

You have used string variables to store character strings and you know that the rules for manipulating string variables or string constants are not the same as those for numeric variables or numeric constants. TONTO BASIC offers a full range of facilities for manipulating character strings effectively. In particular, the concept of string-slicing both extends and simplifies the handling of substrings or slices of a string.

ASSIGNING STRINGS

Storage for string variables is allocated as it is required by a program. For example, the lines:

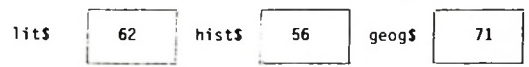
```
10 LET words$ = "LONG"
20 LET words$ = "LONGER"
30 PRINT words$
```

causes the six letter word, LONGER, to be printed. The first line causes space for four letters to be allocated, but this allocation is overruled by the second line which requires space for six characters.

It is, however, possible to dimension (that is, reserve space for) string variables, in which case the maximum length becomes defined, and the variable behaves in every way as an array.

JOINING STRINGS

You may wish to construct records in data processing from a number of sources. Suppose, for example, that you are a teacher and you want to store a set of three marks for each student in Literature, History, and Geography. The marks are held in variables as shown:



As part of student record keeping, you may wish to combine the three string values into one six-character string called marks$. You simply write:

```
LET mark$ = lit$ & hist$ & geog$
```

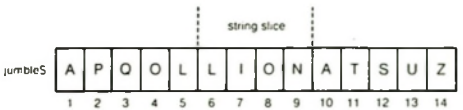
You have created a further variable as shown:

But remember that you are dealing with a character string which happens to be numbers, rather than an actual number.

COPY A STRING SLICE

A string slice is part of a string. It may be anything from a single character to the whole string. In order to identify the string slice you need to know the positions of the required characters.

Suppose you are constructing a children's game in which they have to recognise a word hidden in a jumble of letters. Each letter has an internal number (an index) corresponding to its position in the string. Suppose the whole string is stored in the variable jumble$, and the clue is Big cat.



You can see that the answer is defined by the numbers 6 to 9 which indicate where it is. You can extract the answer as shown:

```
10 jumble$ = "APQOLLIONATSUZ"
20 LET an$ = jumble$(6 TO 9)
30 PRINT an$
```

REPLACE A STRING SLICE

Now suppose that you wish to change the hidden animal into a bull. You can write two extra lines:

```
40 LET jumble$(6 TO 9) = "BULL"
50 PRINT jumble$
```

The output from the whole five-line program is:

```
LION
APQOLBULLATSUZ
```

Note: If you attempt to copy a string into a string-slice which has insufficient length, the assignment may not give the correct result.

If you wish to copy a string into string-slice, it is best to ensure the destination string is long enough by padding it first with spaces (see line 20 in the program below).

```
10 LET subject$ = "ENGLISH MATHS COMPUTING"
20 LET student$ = "                    "
30 LET student$(9 TO 13) = subject$(9 TO 13)
```

We say that BULL is a slice of the string APQOLBULLATSUZ. The defining phrase:

```
(6 TO 9)
```

is called a slicer. It has other uses. Notice how the same notation may be used on both sides of the LET statement. If you want to refer to a single character it is clumsy to write:

```
jumble$(6 TO 6)
```

just to pick out the B so you can write instead:

```
jumble$(6)
```

to refer to a single character.

COERCION

Suppose you have a variable, mark$ holding a record of examination marks. The slice giving the history mark may be extracted and scaled up, perhaps because the history teacher has been too strict in the marking. The following lines extract the history mark:

```
10 LET mark$ = "625671"
20 LET hist$ = mark$(3 TO 4)
```

The problem now is the value 56 of the variable, hist$, is a string of characters not numeric data. If you want to scale it up by multiplying by, say, 1.125, the value of hist$ must be converted to numeric data first. TONTO BASIC does this conversion automatically when you type:

```
30 LET num = 1.125 * hist$
```

Strictly speaking, it is illegal to mix data types in a LET statement. It would be silly to write:

```
LET num = "LION"
```

and you would get an error message if you tried, but if you write:

```
LET num = "56"
```

the system concludes that you want the number 56 to become the value of num. In the program line 30 converts the string "56" to the number 56 and multiplies it by 1.125 giving 63.

Now we should replace the old mark by the new mark but the new mark is still the number 63. Before it can be inserted into the original string it must be converted back to the string "63". Again TONTO BASIC converts the number automatically when you type:

```
40 LET mark$(3 TO 4) = num
50 PRINT mark$
```

The output from the whole program is:

```
626371
```

which shows the history mark increased to 63.

The complete program is:

```
10 LET mark$ = "625671"
20 LET hist$ = mark$(3 TO 4)
30 LET num = 1.125 * hist$
40 LET mark$(3 TO 4) = num
50 PRINT mark$
```

In line 30 a string value was converted into numeric form so that it could be multiplied; in line 40 a number was converted into string form. This converting of data types is known as **type coercion**.

You can write the program more economically now you understand both string-slicing and coercion:

```
10 LET mark$ = "625671"
20 LET mark$(3 TO 4) = 1.125 * mark$(3 TO 4)
30 PRINT mark$
```

Again the output is the same.

If you have worked with other BASICs, you can appreciate the simplicity and power of string-slicing and coercion.

SEARCHING A STRING

You can search a string for a given substring. The following program displays a jumble of letters and invites you to spot the animal.

```
10 REM Animal Spotting
20 LET jumble$ = "SYNDICATE"
30 PRINT jumble$
40 INPUT "What is the animal?"!an$
50 IF (an$ INSTR jumble$) AND an$(1) = "C" AND LEN(an$) = 3
60   PRINT "Correct"
70 ELSE
80   PRINT "Not correct"
90 END IF
```

The expression (an$ INSTR jumble$) returns zero if the guess is incorrect. If the guess is correct, the expression returns the number which is the starting position of the string-slice, in this case 6.

Because the expression can be treated as a logical expression, the position of the string in a successful search can be regarded as true: in an unsuccessful search it can be regarded as false.

OTHER STRING FUNCTIONS

You have already met **LEN** which returns the length (number of characters) of a string.

You may wish to repeat a particular string or character several times. For example, if you wish to output a row of asterisks, rather than actually enter forty asterisks in a **PRINT** statement or organise a loop you can simply write:

```
PRINT FILL$ ("*",40)
```

Finally, it is possible to use the function **CHR$** to convert ASCII codes into string characters. For example:

```
PRINT CHR$(65)
```

outputs A.

COMPARING STRINGS

A great deal of computing is concerned with organising data so it can be searched quickly. Sometimes it is necessary to sort it into alphabetical order. The basis of various sorting processes is the facility for comparing two strings to see which comes first. Because the letters A,B,C... are internally coded as 65,66,67... it is natural to regard the following statements as correct:

```
A is less than B
B is less than C
```

and because internal character by character comparison is automatically provided:

```
CAT is less than DOG
CAN is less than CAT
```

For example, if you write:

```
IF "CAT" < "DOG" THEN PRINT "MEOW"
```

the output is:

```
MEOW
```

Similarly:

```
IF "DOG" > "CAT" THEN PRINT "WOOF"
```

gives the output:

```
WOOF
```

We use the comparison symbols of mathematics for string comparisons as follows:

> Greater than - Case dependent comparison, numbers compared in numerical order

< Less than - Case dependent comparison, numbers compared in numerical order

= Equals - Case dependent comparison, strings must be the same

= = Similar - Strings must be 'almost' the same. Case independent comparison, numbers compared in numerical order

>= Greater than or equal to - Case dependent comparison, numbers compared in numerical order

<= Less than or equal to - Case dependent comparison, numbers compared in numerical order

<> Not equal to. Case dependent comparison, numbers compared in numerical order

All the following logical statement expressions are both permissible and true.

```
"ALF" < "BEN"
"KIT" > "BEN"
"KIT" <= "LEN"
"KIT" >= "KIT"
"PAT" >= "LEN"
"LEN" <= "LEN"
"PAT" <> "PET"
```

So far, comparisons based simply on internal codes make sense, but data is not always conveniently restricted to upper case letters. We would like, for example:

Cat to be less than COT
and K2N to be less than K27N

In the second example above, the 2 is compared with the 27.

A simple character by character comparison based on internal codes would not give these results, so TONTO BASIC allows a more intelligent approach. The following program, with suggested input and the output that results, illustrates the rules for comparison of strings.

```
10 REMark comparisons
20 REPeat comp
30   INPUT "input a string" ! first$
30   INPUT "input another string" ! second$
40   IF first$ < second$ THEN PRINT "Less"
50   IF first$ > second$ THEN PRINT "Greater"
60   IF first$ = second$ THEN PRINT "Equal"
70 END REPeat comp
```

input		output
Cat	COT	Less
CAT	CAT	Equal
PET	PETE	Less
K6	K7	Less
K66	K7	Greater
K12N	K6N	Greater
#	£	Less

EXERCISES ON SECTION 11

1 Place 12 letters, all different, in a string variable and another six letters in a second string variable. Search the first string for each of the six letters in turn saying in each case whether it is found or not found

2 Repeat using single character arrays instead of strings. Place twenty random upper case letters in a string and list those which are repeated

3 Write a program to read a sample of text all in upper case letters. Count the frequency of each letter and print the results:

"GOVERNMENT IS A TRUST, AND THE OFFICERS OF THE GOVERNMENT ARE TRUSTEES; AND BOTH THE TRUST AND THE TRUSTEES ARE CREATED FOR THE BENEFIT OF THE PEOPLE. HENRY CLAY, 1829."

4 Write a program to count the number of words in the following text. A word is recognised because it starts with a letter and is followed by a space, comma, or full stop.

"THE REPORTS OF MY DEATH ARE GREATLY EXAGGERATED. CABLE FROM MARK TWAIN TO THE ASSOCIATED PRESS, LONDON 1896."

5 Rewrite the last program, illustrating the use of logical variables and procedures

12 Screen output

This section describes screen output on the TONTO under two main headings.

The first describes the output of ordinary text under Simple Printing. It explains the methods of displaying messages, text, or numerical output, and introduces the concept of the 'intelligent' space - an example of combining ease of use with very useful effects.

The second section describes the facilities you can use on the screen of the TONTO under screen.

In designing and implementing the screen facilities of the TONTO, we have always remembered that advanced features must not imply incomprehensible keywords and syntax. Each keyword has been carefully chosen to reflect the effect it causes. **WINDOW** an area of the screen; **PAPER** defines the background colour; **INK** determines the colour of what you put on the paper.

SIMPLE PRINTING

The keyword **PRINT** can be followed by a sequence of print items. A print item may be any of:

```
text such as: "This is text"
variables such as: num, word$
expressions such as: 3 * num, day$ & week$
```

Print items may be mixed in any print statement but there must be one or more print separators between each pair. Print separators may be any of:

; Usually no effect on screen. When following the last data item the automatic new line is suppressed

! An intelligent space normally inserts a space between output items. If an item will not fit on the current line it behaves as a new line symbol

, A tabulator causes the output to be tabulated in columns of 8 characters

\ A new line symbol forces a new line

Examples

The numbers 1, 2, 3 are some examples of legitimate print items and are convenient for illustrating the effects of print separators

Statement	Effect
10 PRINT 1,2,3	1 2 3
10 PRINT 1!2!3!	1 2 3
10 PRINT 1\2\3	1 2 3
10 PRINT 1;2;3	123
10 PRINT "This is text"	This is text
10 LET word$ = " " 20 PRINT word$	moves print position
10 LET num = 13 20 PRINT num	13
10 LET an$ = "yes" 20 PRINT "I say"! an$	I say yes
10 PRINT "Sum is"! 4 + 2	Sum is 6

Simple layout

You can position print output anywhere on the screen with the **AT** command.

For example:

```
AT 10,15 : PRINT "This is on the 10th line in column 15"
```

but remember that the top left hand corner is **AT 0,0**

If you read the Keyword Reference Guide you may find it difficult to reconcile the section on **PRINT** with the above description. Two of the difficulties disappear if you understand that:

- Text in quotes, variables and numbers are, strictly speaking, expressions; they are the simplest (degenerate) forms.

- Print separators are strictly classified as print items.

SCREEN

The TONTO provides two screen modes: high resolution mode and low resolution mode. TONTO BASIC always uses the high resolution mode. This mode is characterised by the following:

- The total screen available is 480 pixels wide and 200 pixels deep allowing 20 lines of 80 characters
- A choice of four colours

A pixel is the smallest area of colour which can be displayed.

Colour

On the TONTO monochrome monitor, colours are displayed as different intensities on a grey scale.

You can select a colour using the **PAPER** and **INK** keywords with any of the following values:

Colour Codes

Value	Displayed colour
0	black
1	black
2	dark grey
3	dark grey
4	light grey
5	light grey
6	white
7	white

For example **INK** 3 would give dark grey characters.

Paper

PAPER followed by a number specifies the background. For example:

```
PAPER 2      dark grey
```

The colour is not visible until something else is done, for example, the screen is cleared.

Ink

INK followed by a number specifies the colour for printing characters. For example:

```
INK 7        white ink
```

The ink is changed for all subsequent output.

Cls

CLS means clear the window to the current paper colour -like a teacher cleaning a blackboard, except that it is electronic.

Windows

You can request a window of any size anywhere on the screen, however TONTO BASIC will align it with normal character positions. The device name for a window is:

```
SCR_
```

A window is defined by four constant values, for example:

The following program creates a window with the channel number 5, clears it to dark grey (code 2) and then closes it:

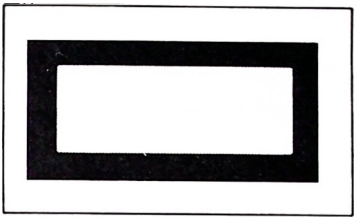
```
10 REMark Create a window
20 OPEN #5, SCR_480x200a18x50
30 PAPER #5,2 : CLS #5
60 CLOSE #5
```

Note that each window can have its own features such as paper, ink, etc. The fact that a window has been opened does not mean that it is the current default window.

You can change the position or shape of an opened window without closing it and reopening it. Try adding two lines to the previous program:

```
40 WINDOW #5,420,70,30,80
50 PAPER #5,4 : CLS #5
```

Re-run the program and you will find a light grey window within the original dark grey one. This light grey window is now the one associated with channel 5, see below.



SPECIAL PRINTING

CSIZE enables you to alter the size of characters. For example:

Csize

```
CSIZE 3,1
```

gives the largest possible characters and:

```
CSIZE 0,0
```

gives the smallest. The first number must be 0, 1, 2 or 3 and determines the width. The second must be 0 or 1 and determines the height. The normal sizes is:

```
CSIZE 0,0    20 lines of 80 characters
```

which is reinstated when a window is changed.

Under

UNDER enables you to underline characters.

```
UNDER 1    underlines all subsequent output in the current
             ink

UNDER 0    switches off underlining
```

EXERCISES ON SECTION 12

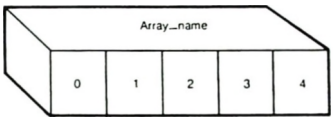
1 Write a program which draws a 'Chess board' of six rows of six squares

2 Place the numbers 1 to 35 in the squares starting at the bottom left and place F for finish in the last square

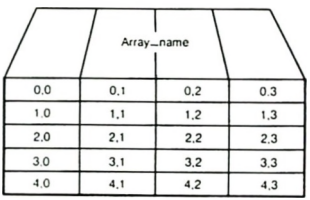
13 Arrays

In section 6 you were introduced to the concept of arrays as a means of handling large amounts of data. This section recaps what you have learned and introduces some more complex uses of various types of array. In its simplest form an array is a variable which contains more than one data item. The variable has a single name and each data item contained in it is referred to by a number.

Arrays can be one-dimensional (like a row of boxes):



or have two or more dimensions (like several rows of boxes). A two-dimensional array can be represented as follows:



0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3
4,0	4,1	4,2	4,3

Array dimensions are explained in greater detail below.

To set up an array you need to declare its name and give its dimensions in a **DIM** statement. To refer to an item in an array, all you need to do is to use the array name and the index of the item (its position in the array).

Obviously the larger the number of data items you have in your program, the greater the advantages of using arrays.

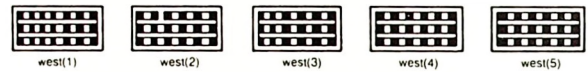
USING ARRAYS

Suppose you are a prison governor and you have a new prison block which is called the West Block. It is ready to receive 50 new prisoners. You need to know which prisoner (known by his number) is in which cell. You could give each cell a name, but it is simpler to give them numbers.

In the following simulation, imagine just 5 prisoners with numbers which you can put in a **DATA** statement:

```
DATA 50, 37, 86, 41, 32
```

Set up an array of variables which share the name west and are distinguished by a number appended in brackets.



It is necessary to declare an array and give its dimensions with a **DIM** statement

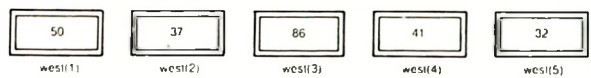
```
DIM west(5)
```

This enables BASIC to allocate the required storage area. After the **DIM** statement has been executed, the five variables can be used.

The convicts can be **READ** from the **DATA** statement into the five array variables:

```
FOR cell = 1 TO 5 : READ west(cell)
```

You can add another **FOR** loop with a **PRINT** statement to prove that the convicts are in the cells



The complete program is shown below:

```
10 REMark Prisoners
20 DIM west(5)
30 FOR cell = 1 TO 5 : READ west(cell)
40 FOR cell = 1 TO 5 : PRINT cell! west(cell)
50 REMark End of Program
60 DATA 50, 37, 86, 41, 32
```

The output from the program is:

```
 1 50
 2 37
 3 86
 4 41
 5 32
```

The numbers 1 to 5 are called indices of the array, *west*. The array, west, is a numeric array consisting of five numeric array elements.

You can replace line 40 by:

```
40 PRINT west
```

This outputs all the elements of the array;

```
 0
50
37
86
41
32
```

The zero at the top of the list appears because the implied index ranges from zero to the declared number. We show later how useful the zero elements in arrays can be.

Note also that when a numeric array is DIMensioned its elements are all given the value zero.

STRING ARRAYS

String arrays are similar to numeric arrays but an extra dimension in the **DIM** statement specifies the maximum length of each string element in the array. Suppose that ten of the top players at Royal Birkdale for the 1982 British Golf Championship are denoted by their first names and placed in **DATA** statements

```
DATA "Tom", "Graham", "Sevvy", "Jack", "Lee"
DATA "Nick", "Bernard", "Ben", "Greg", "Hal"
```

This requires ten different variable names, but if there were a hundred or a thousand players the job would become impossibly tedious. An array is a set of variables designed to cope with problems of this kind. Each variable name consists of two parts:

- a name according to the usual rules for variables
- a numeric part called an index

Write the variable names as:

```
flat$(1), flat$(2), flat$(3)....etc
```

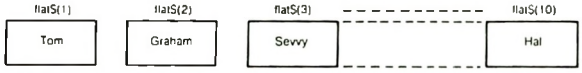
Before you can use the array variables you must tell the system about the array and its dimensions:

```
DIM flat$(10,8)
```

This causes ten variables to be reserved for use in the program. Each string element in the array may have up to eight characters. **DIM** statements should usually be placed all together near the beginning of the program. Once the array has been declared in a **DIM** statement, all the elements of the array can be used. One important advantage is that you can give the numeric part (the index) as a numeric variable. You can write:

```
FOR number = 1 TO 10 : READ flat$(number)
```

which would place the golfers in their flats.



You can refer to the variable in the usual way, but remember to use the right index. Suppose that Tom and Sevvy wished to exchange flats. In computing terms one of them, say Tom, would have to move into a temporary flat to allow Sevvy time to move. You can write:

```
LET temp$ = flat$(1)    : REMark Tom into temporary
LET flat$(1) = flat$(3) : REMark Sevvy into flat$(1)
LET flat$(3) = temp$    : REMark Tom into flat$(3)
```

The following program places the ten golfers in an array named flat$ and prints the names of the occupants with their 'flat numbers' (array indices) to prove that they are in residence. The occupants of flats 1 and 3 then change places. The list of occupants is then printed again to show that the exchange has occurred.

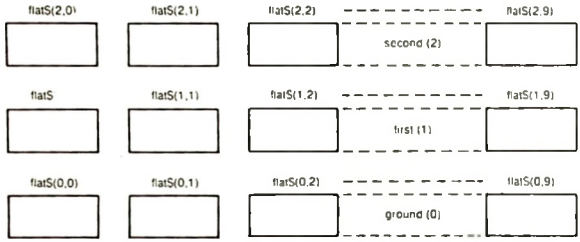
```
 10 REMark Golfers' Flats
 20 DIM flat$(10,8)
 30 FOR number = 1 TO 10 : READ flat$(number)
 40 printlist
 50 exchange
 60 printlist
 70 STOP: REMark End of MAIN Program
 80 DEFine PROCedure printlist
 90   FOR num = 1 TO 10 : PRINT num! flat$(num)
100 END DEFine printlist
110 DEFine PROCedure exchange
120   LET temp$ = flat$(1)
130   LET flat$(1) = flat$(3)
140   LET flat$(3) = temp$
150 END DEFine exchange
160 DATA "Tom", "Graham", "Sevvy", "Jack", "Lee"
170 DATA "Nick", "Bernard", "Ben", "Greg", "Hal"
```

output (line 40)	output (line 60)
1 Tom	1 Sevvy
2 Graham	2 Graham
3 Sevvy	3 Tom
4 Jack	4 Jack
5 Lee	5 Lee
6 Nick	6 Nick
7 Bernhard	7 Bernhard
8 Ben	8 Ben
9 Greg	9 Greg
10 Hal	10 Hal

TWO DIMENSIONAL ARRAYS

Sometimes the nature of a problem suggests two dimensions such as 3 floors of 10 flats rather than just a single row of 30.

Suppose that 20 or more golfers need flats and there is a block of 30 flats divided into three floors of ten flats each. A realistic method of representing the block would be with a two-dimensional array. You can think of the thirty variables as shown below:



Notice how the flats have been arranged to use element 0 (zero) of the array.

Assuming **DATA** statements with 30 names, a suitable way to place the names in the flats is:

```
30 FOR floor = 0 TO 2
40    FOR num = 0 TO 9
50       READ flat$(floor, num)
60    END FOR num
70 END FOR floor
```

You also need a **DIM** statement:

```
20 DIM flat$(2,9,8)
```

which shows that the first index can be from 0 to 2 (floor number) and the second index can be from 0 to 9 (room number). The third number states the maximum number of characters in each array element.

You can add a **PRINT** routine to show that the golfers are in the flats and use letters for the golfer's names to save space by changing the **DIM** statement accordingly.

```
10 REMark 30 Golfers
20 DIM flat$(2,9,1)
30 FOR floor = 0 TO 2
40   FOR num = 0 TO 9
50      READ flat$(floor,num) : REMark Golfers goes in
60   END FOR num
70 END FOR floor
80 REMark End of input
```

```
 90 FOR floor = 0 TO 2
100   PRINT "Floor number"; floor
110   FOR num = 0 TO 9
120     PRINT 'Flat' ! num ! flat$(floor,num)
130   END FOR num
140 END FOR floor
150 DATA "A","B","C","D","E","F","G","H","I","J"
160 DATA "K","L","M","N","O","P","Q","R","S","T"
170 DATA "U","V","W","X","Y","Z","@","£","$","%"
```

The output starts:

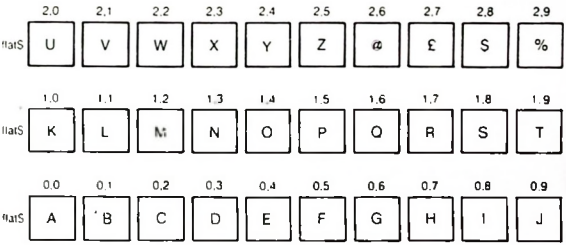
```
Floor number 0
Flat 0A
Flat 1B
Flat 2C
```

and continues, giving the remaining 27 occupants.

ARRAY SLICING

You may find this section hard to read, though it is essentially the same concept as string-slicing. You will probably need string-slicing if you get beyond the learning stage of programming. The need for array-slicing is much rarer and you may wish to omit this section, particularly on a first reading.

We now simplify the golfers-in-flats problem so that we can illustrate the concept of array slicing. The flats are numbered 0 to 9 to keep to single digits and names are single characters for reasons of space.



Given the above values the following examples are array slices:

```
flat$(1,3)       Means a single array element with value N
flat$(1,1 TO 6)  Means the six elements with values L M N
                 O P Q
```

Array Element	Value
flat$(1,1)	L
flat$(1,2)	M
flat$(1,3)	N
flat$(1,4)	O
flat$(1,5)	P
flat$(1,6)	Q

```
flat$(1) Means flat$ (1,0 TO 9) - ten elements
         with values K L M N O P Q R S T
```

In these examples a range of values of an index can be given instead of a single value. If an index is missing completely, the complete range is assumed. In example 3 the second index is missing and it is assumed by the system to be 0 TO 9.

The techniques of array slicing and string slicing are similar, though the latter is more widely applicable.

EXERCISES ON SECTION 13

1 **SORTING**

Place ten numbers in an array by reading from a **DATA** statement. Search the array to find the lowest number. Make this lowest number the value of the first element of a new array. Replace it in the first array with a very large number. Repeat this process making the second lowest number the second value in the new array and so on until you have a sorted array of numbers which should then be printed.

2 **SNAKES AND LADDERS**

Represent a snakes and ladders game with a 100 element numeric array. Each element should contain either:

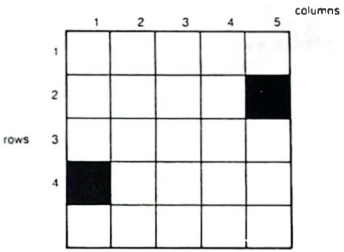
zero

or a number in the range 10 to 90 meaning that a player should transfer to that number by going 'up a ladder' or 'down a snake'

or the digits 1, 2, 3, etc. to denote a particular player's position.

Set up six snakes and six ladders by placing numbers in the array and simulate one 'solo' run by a single player to test the game.

3 **CROSSWORD BLANKS**



Crosswords usually have an odd number of rows or columns in which the black squares have a symmetrical pattern. The pattern is said to have rotational symmetry because rotation through 180 degrees would not change it.

Note that after rotation through 180 degrees, the square in row 4, column 1 could become the square in row 2, column 5. That is row 4, column 1 becomes row (5+1)-4, column (5+1)-1 in a 5 x 5 grid.

Write a program to generate and display a symmetrical pattern of this kind in a 5 x 5 and a 10 x 10 grid.

4 Modify the crossword pattern so that there are no sequences, across or down, of fewer than four white squares.

5 **CARD SHUFFLE**

Cards are denoted by the numbers 1 to 52 stored in an array. They can be converted easily to actual card values when necessary. The cards should be 'shuffled' as follows:

Choose any position in range 1 to 51 (e.g. 17).

Place the card in this position in a temporary store.

Shunt all the cards in positions 52 to 18 down to positions 51 to 17.

Place the chosen card from the temporary store to position 52.

Deal similarly with the ranges 1 to 50, 1 to 49. down to 1 to 2 so that the pack is well shuffled. Output the result of the shuffle.

6 Set up six **DATA** statements each containing a surname, initials and a telephone number (dialling code and local number). Decide on a suitable structure of arrays to store this information and **READ** it into the arrays.

PRINT the data using a separate **FOR** loop and explain how the input format (**DATA**), the internal format (arrays) and output format are not necessarily all the same

14 Program structure

In this section we go over the ground of program structure : loops and decisions or selection. We have tried to present things in as simple a way as possible, but TONTO BASIC is designed to cope properly with the simple and the complex and all levels in between. Some parts of this chapter are difficult and if you are new to programming you may wish to omit parts. The topics covered are:

Loops
Nested loops
Binary decisions
Multiple decisions

The first section, Loops, gets progressively more difficult as we show how TONTO BASIC copes with problems that other languages simply ignore. Skip these parts if you feel so inclined, but the other sections are more straightforward.

LOOPS

This section attempts to illustrate the well-known problems of handling repetition with simulations of some Wild West scenes. The context may be contrived and trivial but it offers a simple basis for discussion and illustrates difficulties which arise across the whole range of programming applications.

Example 1

A bandit is holed up in the Old School House. The sheriff has six bullets in his gun. Simulate the firing of the six shots.

Program 1

```
10 REMark Western FOR
20 FOR bullets = 1 TO 6
30    PRINT "Take aim"
40    PRINT "Fire shot"
50 END FOR bullets
```

Program 2

```
10 REMark Western REPeat
20 LET bullets = 6
30 REPeat bandit
40   PRINT "Take aim"
50   PRINT "Fire shot"
60   LET bullets = bullets - 1
70   IF bullets = 0 THEN EXIT
80 END REPeat bandit
```

Both these programs produce the same output:

```
Take aim
Fire shot
```

is printed six times.

If in each program the 6 is changed to any number down to 1, both programs still work as you would expect. But what if the gun is empty before any shots have been fired?

Example 2

Suppose that someone has secretly taken all the bullets out of the sheriff's gun. What happens if you simply change the 6 to 0 in each program?

Program 1

```
10 REMark Western FOR Zero Case
20 FOR bullets = 1 to 0
30   PRINT "Take aim"
40   PRINT "Fire a shot"
50 END FOR bullets
```

This works correctly. There is no output. The zero case behaves properly in TONTO BASIC.

Program 2

```
10 REMark Western REPeat Fails
20 LET bullets = 0
30 REPeat bandit
40   PRINT "Take aim"
50   PRINT "Fire a shot"
60   LET bullets = bullets -1
70   IF bullets = 0 THEN EXIT bandit
80 END REPeat bandit
```

Program 2 fails in two ways:

1 **Take aim**
Fire a shot

is printed though there were never any bullets

2 By the time the variable, *bullets*, is tested in line 70 it has the value -1 and it never becomes zero afterwards. The program loops indefinitely. You can cure the endless looping by re-writing line 60:

```
70 IF bullets < 1 THEN EXIT bandit
```

There is an inherent fault in the programming which does not allow for the possible zero case. This can be corrected by placing the conditional **EXIT** before the **PRINT** statements, thus:

Program 3

```
10 REMark Western REPeat Zero Case
20 LET bullets = 0
30 REPeat bandit
40   IF bullets = 0 THEN EXIT bandit
50   PRINT "Take aim"
60   PRINT "Fire a shot"
70   LET bullets = bullets -1
80 END REPeat bandit
```

The program now works properly whatever the initial value of bullets as long as it is a positive whole number or zero. Program 2 corresponds to the **REPEAT...UNTIL** loop of some languages. Program 3 corresponds to the **WHILE...ENDWHILE** loop of some languages. However, the **REPeat...END REPeat** with **EXIT** is more flexible than either, or the combination of both.

If you have used other BASICs you may wonder what has happened to the **NEXT** statement. We re-introduce it soon but you will see that the loops have a similar structure and both are named.

FOR *name* =	(opening keyword)	**REPeat** *name* =
statements	(content)	statements
END FOR *name*	(closing keyword)	**END REPeat** *name*

In addition the **REPeat** loop must normally have an **EXIT** amongst the statements or it never ends.

Note also that the **EXIT** statement causes control to go to immediately after the **END** of the loop.

A **NEXT** statement may be placed in a loop. It causes control to go to just after the opening keyword **FOR** or **REPeat**. It should be considered as a kind of opposite to the **EXIT** statement.

NEXT is included here for compatibility with other BASICs. It is unnecessary in well structured programs but its use is shown below:

Example 3

The situation is the same as in example 1. The sheriff has a gun loaded with six bullets and he is to fire at the bandit, but two more conditions apply:

1 If he hits the bandit he stops firing and returns to Dodge City

2 If he runs out of bullets before he hits the bandit, he tells his partner to watch the bandit while he (sheriff) returns to Dodge City

Program 1

```
 10 REMark Western FOR with Epilogue
 20 FOR bullets = 1 TO 6
 30   PRINT "Take aim"
 40   PRINT "Fire a shot"
 50   LET hit = RND(9)
 60   IF hit = 7 THEN EXIT bullets
 70 NEXT bullets
 80   PRINT "Watch Bandit"
 90 END FOR bullets
100 PRINT "Return to Dodge City"
```

In this case, the content between **NEXT** and **END FOR** is a kind of epilogue which is only executed if the **FOR** loop runs its full course. If there is a premature **EXIT**, the epilogue is not executed.

The problem can be better reflected without the use of **NEXT**, thus:

Program 2

```
 10 REMark Western REPeat with Epilogue
 20 LET bullets = 6
 30 REPeat bandit
 40   PRINT "Take aim"
 50   PRINT "Fire a shot"
 60   LET bullets = bullets -1
 70   LET hit = RND(9)
 80   IF (hit=7) OR (bullets = 0) THEN EXIT bandit
 90 END REPeat bandit
100 IF NOT hit THEN PRINT "Watch Bandit"
110 PRINT "Return to Dodge City"
```

This version better reflects the problem by executing lines 30 to 90 until the bandit is shot or the sheriff runs out of bullets. Only when this part is complete does the program make a decision about the required action.

The program works properly as long as the sheriff has at least one bullet at the start. It fails if line 20 reads:

```
20 LET bullets = 0
```

You might think that the sheriff would be a fool to start an enterprise of this kind if he had no bullets at all, and you would be right. We are now discussing how to preserve good structure in the most complex type of situation. We have at least kept the problem context simple; we know what we are trying to do. Complex structural problems usually arise in contexts more difficult than Wild West simulations. But if you really want a solution to the problem which caters for a possible hit, running out of bullets and an epilogue, and also the zero case, add the following lines into the above program and remove line 80:

```
25 hit = 0 : REMark bandit starts off unhurt
35     IF hit OR (bullets = 0) THEN EXIT bandit
```

We can conceive of no more complex type of problem than this with a single loop. TONTO BASIC can easily handle it if you want it to.

NESTED LOOPS

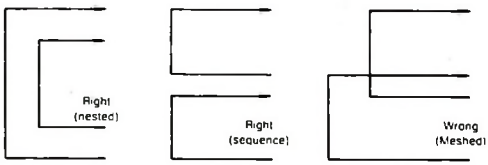
Consider the following **FOR** loop which **PRINT**S a row of crosses in various randomly chosen colour tones (not black).

```
10 REMark Row of crosses
20 PAPER 0 : CLS
30 LET down = 10
40 FOR across = 20 TO 30
50   INK RND(2 TO 7)
60   AT down, across: PRINT "x";
70 END FOR
```

This program prints a row of crosses thus:

```
xxxxxxxxxx
```

If you want to get, say, 11 rows of crosses you must **PRINT** a row for each value of *down* from 5 to 15. But you must always observe the rule that a structure can go completely within another or it can go properly around it. It can also follow in sequence, but it cannot mesh with another structure. Books about programming often show how **FOR** loops can be related with a diagram like this:



In TONTO BASIC the rule applies to all structures. You can solve all problems using them properly. We therefore treat the **FOR** loop as an entity and design a new program:

```
FOR down = 5 TO 15

  FOR across = 20 TO 30
    AT down, across: PRINT "x";
  END FOR across

END FOR down
```

When we translate this into a program, we expect it to work and know what it will do. It prints a rectangle made up of rows of crosses.

```
10 REMark Rows of crosses
20 PAPER 0 : CLS
30 FOR down = 5 TO 15
50   FOR across = 20 TO 30
60     INK RND(2 TO 7)
70     AT down, across : PRINT "x";
80   END FOR across
90 END FOR down
```

Different structures may be nested. Suppose we replace the inner **FOR** loop of the above program by a **REPeat** loop. We terminate the **REPeat** loop when the zero colour code appears for a selection in the range 0 to 7.

```
 10 REMark REPeat within FOR
 20 PAPER 0 : CLS
 30 FOR down = 5 TO 15
 40   AT down, 20
 50   REPeat crosses
 60     LET colour = RND(7)
 70     INK colour
 90     PRINT "x";
100     IF colour < 2 then EXIT crosses
110   END REPeat crosses
120 END FOR down
```

If the program selects colour 0 or 1 as its first option, it will produce no visible output as black ink cannot be seen on black paper!

Much of the wisdom about program control and structure can be expressed in two rules:

1 Construct your program using only the legitimate structures for loops and decision-making

2 Each structure should be properly in sequence or wholly within another larger structure

BINARY DECISIONS

The three types of binary decision can be illustrated easily in terms of what to do when it rains.

Program 1

```
 10 REMark Short form IF
 20 LET rain = RND(0 TO 1)
 30 IF rain THEN PRINT "Open brolly"
```

Program 2

```
 10 REMark Long form IF...END IF
 20 LET rain = RND(0 TO 1)
 30 IF rain THEN
 40   PRINT "Wear coat"
 50   PRINT "Open brolly"
 60   PRINT "Walk fast"
 70 END IF
```

Program 3

```
10 REMark Long form IF...ELSE...END IF
20 LET rain = RND(0 TO 1)
30 IF rain THEN
40   PRINT "Take a bus"
50 ELSE
60   PRINT "Walk"
70 END IF
```

All these are binary decisions. The first two programs are simple: either something happens or it does not. The third is a general binary decision with two distinct possible courses of action, both of which must be defined.

You can omit **THEN** in the long forms if you wish. In the short form you can substitute : for **THEN**.

Example

Consider a more complex example in which it seems natural to nest binary decisions. This type of nesting can be confusing and you should only do it if it seems the most natural thing to do. Careful attention to layout, particularly indenting, is especially important.

Analyse a piece of text to count the number of vowels, consonants and other characters. Ignore spaces. For simplicity the text is all upper case.

```
"COMPUTER HISTORY WAS MADE IN 1984"
```

Program design

```
Read in the data
 FOR each character
   IF letter THEN
      IF vowel
         increase vowel count
      ELSE
        increase consonant count
      END IF
   ELSE
     IF not space THEN increase other count
   END IF
 END FOR
 PRINT results
```

Program

```
 10 REMark Character Counts
 20 READ text$
 30 LET vowels = 0 : cons = 0 : others = 0
 40 FOR num = 1 TO LEN(text$)
 50   LET ch$ = text$(num)
 60   IF ch$ >= "A" AND ch$ <= "Z"
 70     IF ch$ INSTR "AEIOU"
 80       LET vowels = vowels + 1
 90     ELSE
100       LET cons = cons + 1
110     END IF
120   ELSE
130     IF ch$ <> " " THEN others = others + 1
140   END IF
150 END FOR num
160 PRINT "Vowel count is"! vowels
170 PRINT "Consonant count is"! cons
180 PRINT "Other count is"! others
190 DATA "COMPUTER HISTORY WAS MADE IN 1984"
```

Results in the output:

```
Vowel count is 9
Consonant count is 15
Other count is 4
```

MULTIPLE DECISIONS
The SELect statement

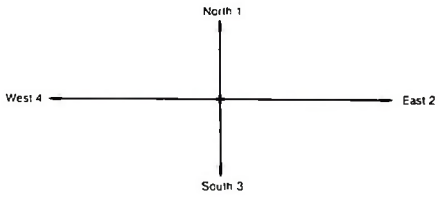
Where there are three or more possible actions and none is dependent on a previous choice, the natural structure to use is SELect which enables selection from any number of possibilities.

Example

A magic snake grows without limit by adding a section to its front. Each section may be up to twenty units long and may be a new colour, or it may remain the same. Each new section must grow in one of the directions North, South, East and West. The snake starts from the centre of the window.

Method

At any time while the snake is still on the screen, you choose a random length and ink colour. The direction may be selected by a number 1, 2, 3 or 4 as shown:



Program design

```
Select PAPER
Set snake to centre of window
REPeat
  Choose direction, colour, length of growth
  FOR unit = 1 to growth
    Make snake grow, north, south, east or west
    If snake is off window then exit
  END FOR
END REPeat
PRINT end message
```

Program

```
100 REMark magic snake
110 PAPER 0 : CLS
120 LET across = 39 : up = 10
130 REPeat snake
140   LET direction = RND(1 TO 4) : colour = RND(2 TO 7)
150   LET growth = RND(1 TO 6)
160   INK colour
170   FOR unit = 1 TO growth
180     SELect ON direction
190       ON direction = 1
200         LET up = up + 1
210       ON direction = 2
220         LET across = across + 1
230       ON direction = 3
240         LET up = up - 1
```

```
250       ON direction = 4
260         LET across = across - 1
270       END SELect
280       IF across<1 OR across>78 OR up<1 OR up>18 THEN EXIT
                                                        snake
290       AT 19-up,across : PRINT CHR$(127);
300     END FOR unit
310 END REPeat snake
320 PRINT "Snake off edge"
```

The syntax of the SELect ON structure also allows for the possibility of selecting on a list of values such as

```
5, 6, 8, 10 TO 13
```

It is also possible to allow for an action to be executed if none of the stated values is found. The full structure is of the form given below.

Long form

```
SELect ON num
ON num = list of values
  statements
ON num = list of values
  statements
  -
  -
  -
  -
ON num = REMAINDER
  statements
END SELect
```

where *num* is any numeric variable and the REMAINDER clause is optional.

Short form

There is a short form of the SELect structure. For example:

```
10 INPUT num
20 SELect ON num = 0 TO 9 : PRINT "number is a single
                                   digit"
```

performs as you would expect.

EXERCISES ON SECTION 14

1 Store ten numbers in an array and perform a 'bubble sort'. This is done by comparing the first pair and exchanging, if necessary, the second pair (second and third numbers), up to the ninth pair (ninth and tenth numbers). The first run of nine comparisons and possible exchanges guarantees that the highest number will reach its correct position. Another eight runs will guarantee eight more correct positions, leaving only the lowest number which must be in the only (correct) position left. The simplest form of 'bubble sort' of ten numbers requires nine runs of nine comparisons

2 Consider ways of speeding up bubblesort, but do not expect that it will ever be very efficient

3 An auctioneer wishes to sell an old clock and he has instructions to invite a first bid of £50. If no-one bids he can come down to £40, £30, £20, but no lower, in an effort to start the bidding. If no-one bids, the clock is withdrawn from the sale. When bidding starts, he takes only £5 increases until the final bid is made. If the final bid is £35 (the 'reserve price') or more, the clock is sold. Otherwise it is withdrawn.

Simulate the auction using the equivalent of a six-sided die throw to start the bidding. A 'six' at any of the starting prices will start the bidding off.

When the bidding has started there should be a three out of four chance of a higher bid at each invitation

4 In a wild west shoot-out, the Sheriff has no ammunition and wishes to arrest a gunman camped in a forest. He rides amongst the trees tempting the gunman to fire. He hopes that when six shots have been fired he can rush in and overpower the gunman as he tries to reload. Simulate the encounter giving the gunman a one-twentieth chance of hitting the Sheriff with each shot. If the Sheriff has not been hit after six shots, he has a 75% chance of overpowering the gunman

5 The sheriff's instructions to his Deputy are:

"If the gun is empty, re-load it, and if it ain't then keep on firing until you hit the bandit or he surrenders. If Mexico Pete turns up, get out fast."

Write a program which caters properly for all these situations:

Whatever happens, sheriff returns to Dodge City
If Mexico Pete turns up, return immediately
If the gun is empty, reload it
If the gun is not empty, ask the bandit to surrender
If the bandit surrenders, arrest him
If he doesn't surrender, fire a shot
If the bandit is hit, arrest him and fix his wound

Assume an unlimited supply of ammunition. Use a simulated 'twenty-sided die' and let a seven mean 'surrender' and a 'thirteen' mean the bandit is hit.

15 Procedures and functions

SUMMARY

The first part of this section explains the more straightforward features of TONTO BASIC's procedures and functions. This is done with very simple examples so that you can understand the working of each feature as it is described. Though the examples are simple and contrived, you will appreciate that, once understood, the ideas can be applied in more complex situations where they really matter.

After the first part, there is a discussion which attempts to explain 'Why procedures?'. If you understand - more or less - up to that point you will be doing well, and you should be able to use procedures and functions with increasing effectiveness.

TONTO BASIC first allows you to do the simpler things in simple ways and then offers you more if you want it. Extra facilities and some technical matters are explained in the second part of this section but you could omit these, certainly at a first reading, and still be in a stronger position than most users of older types of BASIC.

VALUE PARAMETERS

You have seen in previous sections how a value can be passed to a procedure. Here is another example

Example 1

In Chan's Chinese Take-Away there are just six items on the menu.

Rice Dishes	Sweets
1 prawns	4 ice-cream
2 chicken	5 fritter
3 special	6 lychees

Chan has a simple way of computing prices. He works in pence and the prices are:

for a rice dish 300 + 10 times menu number
for a sweet 12 times menu number

Thus a customer who ate special rice and an ice-cream would pay:

```
300 + 10 * 3 + 12 * 4 = 378 pence
```

A procedure, item, accepts a menu number as a value parameter and prints the cost.

Program

```
10 REMark Cost of Dish
20 item 3
30 item 4
40 DEFine PROCedure item(num)
50   IF num <= 3 THEN LET price = 300 + 10 * num
60   IF num >= 4 THEN LET price = 12 * num
70   PRINT !price!
80 END DEFine
```

Output: 330 48

In the main program, actual parameters 3 and 4 are used. The procedure definition has a formal parameter, *num*, which takes the value passed to it from the main program. Note that the formal parameters must be in brackets, but that actual parameters need not be.

Example 2

Now suppose the working variable, price, was also used in the main program, meaning something else, say the price of a glass of lager, 70p. The following program fails to give the desired result:

Program

```
 10 REMark Global price
 20 LET price = 70
 30 item 3
 40 item 4
 50 PRINT !price!
 60 DEFine PROCedure item(num)
 70   IF num <= 3 THEN price = 300 + 10 * num
 80   IF num >= 4 THEN price = 12 * num
 90   PRINT !price!
100 END DEFine
```

Output: 330 48 48

The price of the lager has been altered by the procedure. We say that the variable, price, is global because it can be used anywhere in the program.

Example 3

Make the procedure variable, price, LOCAL to the procedure. This means that TONTO BASIC treats it as a special variable accessible only within the procedure. The variable, price, in the main program will be a different thing even though it has the same name.

Program

```
 10 REMark LOCAL price
 20 LET price = 70
 30 item 3
 40 item 4
 50 PRINT !price!
 60 DEFine PROCedure item(num)
 70   LOCAL price
 80   IF num <= 3 THEN LET price = 300 + 10*num
 90   IF num >= 4 THEN LET price = 12*num
100   PRINT !price!
110 END DEFine
```

Output: 330 48 70

This time everything works properly. Line 70 causes the procedure variable, price to be internally marked as belonging only to the procedure, item. The other variable, price, is not affected. You can see that local variables are useful things.

Example 4

Local variables are so useful that we automatically make procedure formal parameters local. Though we have not mentioned it before parameters such as num in the above programs cannot interfere with main program variables. To prove this we drop the **LOCAL** statement from the above program and use num for the price of lager. Because num in the procedure is local, everything works.

Program

```
 10 REMark LOCAL parameter
 20 LET num = 70
 30 item 3
 40 item 4
 50 PRINT num
 60 DEFine PROCedure item(num)
 70   IF num <= 3 THEN LET price = 300 + 10*num
 80   IF num >= 4 THEN LET price = 12*num
 90   PRINT !price!
100 END DEFine
```

Output: 330 48 70

VARIABLE PARAMETERS

So far we have only used procedure parameters for passing values to the procedure. But suppose the main program wants the cost of an item to be passed back so that it can compute the total bill. We do this easily by providing another parameter in the procedure call. This must be a variable, because it has to receive a value from the procedure. We therefore call it a **variable parameter** and it must be matched by a corresponding variable parameter in the procedure definition.

Example

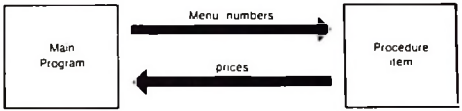
Use actual variable parameters, *cost_1* and *cost_2* to receive the values of the variable price from the procedure. Make the main program compute and print the total bill.

Program

```
 10 REMark Variable parameter
 20 LET num = 70
 30 item 3,cost_1
 40 item 4,cost_2
 50 LET bill = num + cost_1 + cost_2
 60 PRINT bill
 70 DEFine PROCedure item(num, price)
 80   IF num <= 3 THEN LET price = 300 + 10*num
 90   IF num >= 4 THEN LET price = 12*num
100 END DEFine
```

Output: 448

The parameters num and price are both automatically local so there can be no problems. The diagram shows how information passes from main program to procedure and back:



That is enough about procedures and parameters for the present.

FUNCTIONS

You already know how a system function works. For example the function:

```
SQRT(9)
```

computes the value, 3, which is the square root of 9. We say the function returns the value 3. A function, like a procedure, can have one or more parameters, but the distinguishing feature of a function is that it returns exactly one value. This means that you can use it in expressions that you already have. You can type:

```
PRINT 2*SQRT(9)
```

and get the output 6. Thus a function behaves like a procedure, with one or more value parameters and exactly one variable parameter holding the returned value; that variable parameter is referenced using the function name itself.

The parameters need not be numeric.

```
LEN("string")
```

has a string argument but it returns the numeric value 6.

Example

Re-write the program of the last section which uses price as a variable parameter. Let *price* be the name of the function.

Notice the simplification in the calling of functions as compared with procedure calls.

Program

```
10 REMark FuNction with RETURN
20 LET num = 70
30 LET bill = num + price(3) + price(4)
40 PRINT bill
50 DEFine FuNction price(num)
60    IF num <= 3 THEN RETURN 300 + 10* num
70    IF num >= 4 THEN RETURN 12* num
80 END DEFine
```

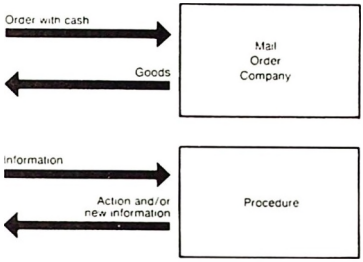
Output: 448

The abbreviation for **FuNction** is **FN** for compatibility with other BASICs. This accounts for its odd form in programs.

WHY PROCEDURES?

The ultimate concept of a procedure is that it should be a black box which receives specific information from 'outside', and performs certain operations which may include sending specific information back to the 'outside'. The 'outside' may be the main program or another procedure.

The term *black box* implies that its internal workings are not important; you only think about what goes in and what comes out. If, for example, a procedure uses a variable, *count*, and changes its value, that might affect a variable of the same name in the main program. Think of a mail order company. You send them an order and cash; they send you goods. Information is sent to a procedure and it sends back action and/or new information.



You do not want the mail order company to use your name and address, or other information, for other purposes. That would be an unwanted side-effect. Similarly, you do not want a procedure to cause unplanned changes to values of variables used in the main program.

Of course you could make sure that there are no double uses of variable names in a program. That will work up to a point but we have shown in this chapter how to avoid trouble even if you do not know what variables are used outside a procedure.

A second aim in using procedures is to make a program modular. Rather than have one long main program, you can break the job down into what Seymour Papert - the inventor of LOGO - calls Mind-sized bites. These are the procedures, each one small enough to understand and control easily. They are linked together by procedure calls in a sequence or hierarchy.

A third aim is to avoid writing the same code twice. Write it once as a procedure and call it twice if necessary.

We give below another example which shows how procedures make a program modular.

Example

An order is placed for six dishes at Chan's Take Away, where the menu is:

Item Number	Dish	Price
1	Prawns	3.50
2	Chicken	2.80
3	Special	3.30

Write procedures for the following tasks.

1 Set up two three-element arrays showing menu, dishes and prices. Use a **DATA** statement

2 Simulate an order for six randomly chosen dishes using a procedure, *choose*, and make a tally of the number of times each dish is chosen

3 Pass the three numbers to a procedure, *waiter*, which passes back the cost of the order to the main program using a parameter *cost*. Procedure *waiter* calls two other procedures, *compute* and *cook*, which compute the cost and simulate cooking.

4 The procedure, *cook*, simply prints the number required and the name of each dish

The main program should call procedures as necessary, get the total cost from procedure *waiter*, add 15% VAT and print the amount of the total bill.

Program design

This program illustrates parameter passing in a fairly complex way, and we explain the program step by step before putting it together.

```
10 REMark Procedures
20 DIM item$(3,7), price(3), dish(3)
30 LET vat = 1 + 15/100
40 set-up
-
-
```

```
100 DEFine PROCedure set_up
110   FOR k = 1 TO 3
120     READ item$(k)
130     READ price(k)
140   END FOR k
150 END DEFine
-
-
-
370 DATA "Prawns", 3.5, "Chicken", 2.8, "Special", 3.3
```

The names of menu items and their prices are placed in the arrays *item$* and *price*.

The next step is to choose a menu number for each of the six customers. The tally of the number of each dish required is kept in the array *dish*.

```
 50 choose dish
-
-
-
160 DEFine PROCedure choose(dish)
170   FOR pick = 1 TO 6
180     LET number = RND(1 TO 3)
190     LET dish(number) = dish(number) + 1
200   END FOR pick
210 END DEFine
```

Note that the identifier *dish* is

an array in the main program
a formal parameter which is local to the procedure *choose*

In this case, the actual parameter *dish* has the effect of making the identifier refer to the same thing. However, if the actual parameter in line 50 were changed to *price*, the three values passed back from the procedure *choose* would be reflected in array *price*, not *dish*

```
 60 waiter dish, bill
-
-
-
-
220 DEFine PROCedure waiter(dish, cost)
230   compute dish, cost
240   cook dish
250 END DEFine
```

The waiter passes the information about the number of each dish required to the procedure, *compute*, which computes the cost and returns it.

```
260 DEFine PROCedure compute(dish, total)
270   LET total = 0
280   FOR k = 1 TO 3
290     LET total = total + dish(k)*price(k)
300   END FOR k
310 END DEFine
```

The waiter also passes information to the cook who simply prints the number required for each menu item.

```
320 DEFine PROCedure cook(dish)
330   FOR c = 1 TO 3
340     PRINT ! dish(c) ! item(c) !
350   END FOR c
360 END DEFine
```

Again, the array, *dish* in the procedure *cook* is local. It receives the information which the procedure uses in its **PRINT** statement.

The complete program is listed below.

Program

```
 10 REMark Procedures
 20 DIM item$(3,7), price(3), dish(3)
 25 REMark *** PROGRAM ***
 30 LET vat = 1 + 15/100
 40 set_up
 50 choose dish
 60 waiter dish, bill
 80 LET bill = bill * vat
 90 PRINT "Total cost is £" ; bill
 95 REMark *** PROCEDURE DEFINITIONS ***
100 DEFine PROCedure set_up
110   FOR k = 1 TO 3
120     READ item$(k)
130     READ price(k)
140   END FOR k
150 END DEFine
160 DEFine PROCedure choose(dish)
170   FOR pick = 1 TO 6
180     LET number = RND(1 TO 3)
190     LET dish(number) = dish(number) + 1
200   END FOR pick
```

```
210 END DEFine
220 DEFine PROCedure waiter(dish, cost)
230   compute dish, cost
240   cook dish
250 END DEFine
260 DEFine PROCedure compute(dish, total)
270   LET total = 0
280   FOR k = 1 TO 3
290     LET total = total + dish(k)*price(k)
300   END FOR k
310 END DEFine
320 DEFine PROCedure cook(dish)
330   FOR c = 1 TO 3
340   PRINT ! dish(c) ! item$(c)
350   END FOR c
360 END DEFine
365 REMark *** PROGRAM DATA ***
370 DATA "Prawns", 3.5, "Chicken", 2.8, "Special", 3.3
```

The output depends on the random choice of dishes, but the following choice illustrates the pattern and gives a sample output:

```
3 prawns
1 Chicken
2 Special

Total cost is £21.89
```

COMMENT

Obviously the use of procedures and parameters in such a simple program is not necessary, but imagine that each sub-task might be much more complex. In such a situation the use of procedures would allow a modular build-up of the program with testing at each stage. The above example merely illustrates the main notations and relationships of procedures.

Similarly, the next example illustrates the use of functions.

Note that in the previous example the procedures *waiter* and *compute* both return exactly one value. Re-write the procedures as functions and show any other changes necessary as a consequence, as follows:

```
DEFine FuNction waiter(dish)
  cook dish
  RETURN compute dish
END DEFine
```

```
DEFine FuNction compute(dish)
  LET total = 0
  FOR k = 1 TO 3
    LET total = total + dish(k)*price(k)
  END FOR k
END DEFine
```

The function call to *waiter* also takes a different form

```
LET bill = waiter(dish)
```

This program works as before. Notice that there are fewer parameters, though the program structure is similar. That is because the function names are also serving as parameters returning information to the source of the function call.

Example

All the variables used as formal parameters in procedures or functions are 'safe' because they are automatically local. Which variables used in the procedures or functions are not local? What additional statements would be needed to make them local?

Program changes

The variables *k*, *pick*, *c* and *number* are not local. The necessary changes to make them so are:

```
105 LOCal k
165 LOCal pick, number
265 LOCal k
325 LOCal c
```

TYPELESS PARAMETERS

Formal parameters do not have any type. (We have not mentioned this fact before, because you can work perfectly well without this knowledge!) They may look as though they have a type and you may prefer that a variable which handles numbers should look numeric, and a variable which handles strings should look as though it does. But however you write your parameters they are typeless. To prove it, try the following program.

Program

```
10 REMark Number or word
20 waiter 2
30 waiter "Chicken"
40 DEFine PROCedure waiter(item)
50   PRINT item
60 END DEFine
```

Output: 2 Chicken

The type of the parameter is determined only when the procedure is called and an actual parameter arrives.

SCOPE OF VARIABLES

Consider the following program and try to consider what two numbers will be output.

```
 10 REMark scope
 20 LET number = 1
 30 test
 40 DEFine PROCedure test
 50   LOCAL number
 60   LET number = 2
 70   PRINT number
 80   try
 90 END DEFine
100 DEFine PROCedure try
110   PRINT number
120 END DEFine
```

Obviously the first number to be printed is 2, but is the variable *number* in line 100 global?

The answer is that the value of *number* in line 60 is carried into the procedure *try*. A variable which is local to a procedure will be the same variable in a second procedure called by the first.

Equally, if the procedure *try* is called by the main program the variable *number* will be the same number in both the main program and procedure *try*. The implications may seem strange at first but they are logical.

1 The variable *number* in line 20 is global

2 The variable *number* in procedure *try* is definitely local to the procedure

3 The variable *number* in procedure *try* belongs to the part of the program which was the last call to it

We have covered many concepts in this section because TONTO BASIC functions and procedures are very powerful. However, you should not expect to use all these features immediately. Use procedures and functions in simple ways at first. They can be very effective and the power is there if you need it.

EXERCISES ON SECTION 15

1 Six employees are identified by their surnames only. Each employee has a particular pension fund rate expressed as a percentage. The following data represents the total salaries and pension fund rates of the six employees

Benson	13,800	6.25
Hanson	8,700	6.00
Johnson	10,300	6.25
Robson	15,000	7.00
Thomson	6,200	6.00
Watson	5,100	5.75

Write procedures to:

input the data into arrays
compute the annual pension fund contributions
output the lists of names and computed contributions

Link the procedures with a main program calling them in sequence

2 Write a function *pick* with two arguments *range* and *miss*. The function should return a random whole number in the given range but it should not be the value of *miss*

Use the function in a program which chooses a random **PAPER** colour and then prints random letters in random **INK** colours, so that none is the same colour as the **PAPER**

3 Re-write the solution to exercise 1, so that a function *pension* takes salary and contribution rate as arguments and returns the computed pension contribution. Use two procedures, one to input the data and one to output the required information using the function *pension*

4 Write the following:

a procedure which sets up a 'pack of cards'
a procedure which shuffles the cards
a function which takes a number as an argument and
returns a string value describing the card
a procedure which deals and displays four poker hands
of five cards each
a main program which calls the above procedures

(see section 16 for discussion of a similar problem)

16 Some techniques

In this final section we present some applications of concepts and facilities already discussed and show how some further ideas may be applied.

SIMULATION OF CARD PLAYING

It is easy to store and manipulate 'playing cards' by representing them with the numbers 1 to 52. This is how you might convert such a number to the equivalent card. Suppose, for example, that the number 29 appears. You may decide that:

```
cards  1-13 are hearts
cards 14-26 are clubs
cards 27-39 are diamonds
cards 40-52 are spades
```

and you will know that 29 means that you have a diamond. You can program the TONTO to do this with:

```
LET suit = (card-1) div 13
```

This produces a value in the range 0 to 3 which you can use to cause the appropriate suit to be printed. The original value can be reduced to the range 1 to 13 by writing:

```
LET value = card MOD 13
IF value = 0 THEN LET value = 13
```

The numbers 1 to 13 can be made to print Ace, 2, 3... Jack, Queen, King, or, if you prefer it, such phrases as "two of hearts" can be printed. The following program prints the name of the card corresponding to your input number:

Program

```
 10 REMark Cards
 20 DIM suitname$(4,8),cardval$(13,5),
 30 LET f$ = "of"
 40 set_up
 50 REPeat cards
 60   INPUT "Enter a card number 1-52:" ! card
 70   IF card <1 OR card> 52 THEN EXIT cards
 80   LET suit = (card-1) DIV 13
 90   LET value = card MOD 13
100   IF value = 0 THEN LET value = 13
120   PRINT cardval$(value) ! f$ ! suitname$(suit)
130 END REPeat cards
```

```
140 DEFine PROCedure set up
150   FOR s = 1 TO 4 : READ suitname$(s)
160   FOR v = 1 TO 13 : READ cardval$(v)
170 END DEFine
180 DATA "hearts","clubs","diamonds","spades"
190 DATA "Ace","Two","Three","Four","Five","Six","Seven"
200 DATA "Eight","Nine","Ten","Jack","Queen","King"
```

A sample input with corresponding output is given below:

Input	Output
13	King of hearts
49	Ten of spades
27	Ace of diamonds

COMMENT

Notice the use of **DATA** statements to hold a permanent file of data which the program always uses. The other data, which changes each time the program runs, is entered through an **INPUT** statement. If the input data is known before running the program, it would be equally correct to use another **READ** and more **DATA** statements. This gives better control.

SEQUENTIAL DATA FILES

The following program establishes a file of one hundred numbers.

Numeric file

```
10 REMark Number File
20 OPEN_NEW #6,MDV1_numbers
30 FOR num = 1 TO 100
40   PRINT #6,num
50 END FOR num
60 CLOSE #6
```

You can get a view of the file - without any special formatting - by copying from microdrive to screen:

```
COPY MDV1_numbers to SCR_
```

You can also use the following program to read the file and display its records on the screen.

```
10 REMark Read File
20 OPEN_IN #6,MDV1_numbers
30 FOR num = 1 TO 100
40   INPUT #6,item
50   PRINT ! item !
60 END FOR num
70 CLOSE #6
```

If you wish, you can alter the program to get the output in a different form.

Character file

In a similar fashion, the following programs set up a file of one hundred randomly selected letters and read them back.

Program 1

```
10 REMark Letter File
20 OPEN_NEW #6,MDV1_chfile
30 FOR num = 1 TO 100
40   LET ch$ = CHR$(RND(65 TO 90))
50   PRINT #6,ch$
60 END FOR num
70 CLOSE #6
```

Program 2

```
10 REMark Get Letters
20 OPEN_IN #6,MDV1_chfile
30 FOR num = 1 TO 100
40   INPUT #6,item$
50   PRINT ! item$ !
60 END FOR num
70 CLOSE #6
```

SETTING UP A DATA FILE

Suppose that you wish to set up a simple file of names and telephone numbers.

```
RON      678462
GEOFF    986487
ZOE      249386
BEN      584621
MEG      482349
CATH     438975
WENDY    982387
```

You can use the following program to do it:

```
10 REMark Phone numbers
20 OPEN_NEW #6,MDV1_phone
30 FOR record = 1 TO 7
40   INPUT name$,num$
50   PRINT #6,name$,num$
60 END FOR record
70 CLOSE #6
```

Type **RUN** and enter a name followed by the **ENTER** key and a number followed by the **ENTER** key. Repeat this seven times.

READ A FILE

You need to be certain that the file exists in a correct form so you should read it back from a microdrive and display it on the screen. You can do this easily using the following program:

```
10 REMark Read Phone Numbers
20 OPEN_IN#5,MDV1_phone
30 FOR record = 1 TO 7
40   INPUT#5,rec$
50   PRINT rec$
60 END FOR record
70 CLOSE#5
```

The data is printed and each pair of fields is held in the variable, *rec$*, before being printed on the screen. You have the opportunity to manipulate it into any desired form.

Note that more than one string may be used at the file creation stage with **INPUT** and **PRINT**, but the whole record so created may be retrieved from the microdrive file with a single string variable (*rec$* in the above example).

AN INSERTION SORT

Example

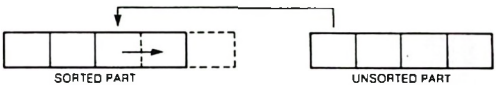
Suppose you want to write a program to sort the following colours into alphabetical order:

black blue red magenta green cyan yellow white

you can use an insertion sort.

Method

Place the eight colours in an array, *colour$*, which is divided into two parts:



You take the leftmost item of the unsorted part and compare it with each item - from right to left - in the sorted part until you find its right place. As you compare you shuffle the sorted items to the right so that when you find the right place to insert you can do so immediately, without further shuffling.

Suppose you have reached the point where four items are sorted and you now focus on green, the leftmost item in the unsorted part.

1	2	3	4	5	6	7	8
black	**blue**	**magenta**	**red**	**green**	**cyan**	**yellow**	**white**
	sorted part				**unsorted part**		

1 Place green in the variable, *comp$*, and set a variable, *p*, to 5

2 The variable, *p*, will eventually indicate where you think green should go. When you know that green should move left, you decrease the value of *p*

3 Compare green with red. If green is greater than (nearer to Z), or equal to, red, you exit and green stays where it is

Otherwise you copy red into position 5 and decrease the value of *p* thus:

1	2	3	4	5	6	7	8
black	blue	magenta	red	red	cyan	yellow	white
			↑				

4 Now repeat the process but this time you are comparing green with magenta and you get:

1	2	3	4	5	6	7	8
black	blue	magenta	magenta	red	cyan	yellow	white
		↑					

5 Finally you move left again, comparing green with blue. This time there is no need to move or change anything. You exit from the loop and place green in position 3. You are then ready to focus on the sixth item, cyan

1	2	3	4	5	6	7	8
black	blue	green	magenta	red	cyan	yellow	white
					↑		

Problem analysis

1 First store the colours in an array colour$(8) and use:

comp$	the current colour being compared
p	to point at the position where you think the colour in *comp$* might go

2 A **FOR** loop will focus attention on positions 2 to 8 in turn (a single item is already sorted)

3 A **REPeat** loop will allow comparisons until you find where the *comp$* value actually goes

```
REPeat compare
  IF comp$ need go no further left EXIT
  copy a colour into the position on its right
  and decrease p
END REPeat compare
```

4 After **EXIT** from the **REPEAT** loop the colour in *comp$* is placed in position *p* and the **FOR** loop continues

Program design

```
1     Declare array colour$
2     Read colours into the array
3     FOR item = 2 TO 8
        LET p = item
        LET comp$ = colour$(p)
        REPeat compare
          IF comp$ >= colour$(p-1): EXIT compare
          LET colour$(p) = colour$(p-1)
          LET p = p-1
        END REPeat compare
        LET colour$(p) = comp$
      END FOR item
4     PRINT sorted array colour$
5     DATA
```

Further testing reveals a fault. It arises very easily if you have data in which the first item is not in its correct position at the start. A simple change of initial data to:

```
red black blue magenta green cyan yellow white
```

reveals the problem. You compare black with red and decrease *p* to the value, 1. You come round again and try to compare black with a variable *colour$(p-1)*, which is *colour$(0)* which has not been assigned a colour.

This is a well-known problem in computing and the solution is to "post a sentinel" on the end of the array. Just before entering the REPeat loop you need:

```
LET colour$(0) = comp$
```

Fortunately, TOMTO BASIC allows a zero index, otherwise the problem would have to be solved at the expense of readability.

Modified program

```
 10 REMark Insertion Sort
 20 DIM colour$(8,7)
 30 FOR item = 1 TO 8 : READ colour$(item)
 40 FOR item = 2 TO 8
 50   LET p = item
 60   LET comp$ = colour$(p)
 70   LET colour$(0) = comp$
 80   REPeat compare
 90     IF comp$>=colour$(p-1) : EXIT compare
100     LET colour$(p) = colour$(p-1)
```

```
120     LET p = p-1
130   END REPeat compare
140   LET colour$(p) = comp$
150 END FOR item
160 PRINT "Sorted..." ! colour$
170 DATA "black","blue","magenta","red"
180 DATA "green","cyan","yellow","white"
```

COMMENT

1 In fact the previous program will work without a sentinel as TONTO BASIC will have set *colour$(0)* to an empty string. You should only take advantage of such good fortune when you are a very experienced programmer

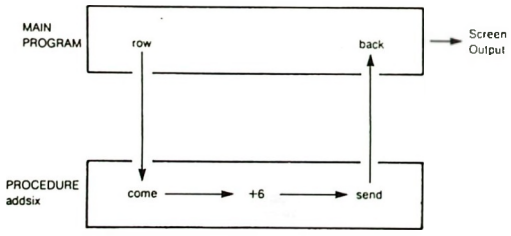
2 An insertion sort is not particularly fast, but it can be useful for adding a few items to an already sorted list. To do a complete resorting it is sometimes convenient to allow modest amounts of time frequently to keep items in order rather than a substantial amount of time less frequently

ARRAY PARAMETERS

In the following program we illustrate the passing of complete arrays between main program and procedure. The data passes in both directions.

In line 40 the array, *row*, holding the numbers 1, 2, 3 is passed to the procedure, *addsix*. The parameter, *come*, receives the incoming data and the procedure adds six to each element. The array parameter, *send*, at this point holds the numbers 7, 8, 9.

These numbers are passed back to the main program to become the values of array, *back*. The values are printed to prove that the data has changed as required.



Program

```
10 REMark pass Arrays
20 DIM row(3),back(3)
30 FOR k = 1 TO 3 : LET row(k) = k
40 addsix row, back
50 FOR k = 1 TO 3 : PRINT back(k)
60 DEFine PROCedure addsix(come,send)
70   FOR k = 1 TO 3 : LET send(k)=come(k)+6
80 END DEFine
```

Output: 7 8 9

The following procedure receives an array containing data to be sorted; the zero element contains the number of items. Note that it does not matter whether the array is numeric or string. The principle of coercion will change string to numeric data if necessary.

A second point of interest is that the array element, *come(0)*, is used for two purposes:

```
it carries the number of items to be sorted
it is used to hold the item currently being placed
```

```
800 DEFine PROCedure sort(come, send)
810   LET num = come(0)
820   FOR item = 2 TO num
830     LET p = item
840     LET come(0) = come(p)
850     REPeat compare
860       IF come(0)>= come(p-1) : EXIT compare
870       LET come(p) = come(p-1)
880       LET p = p-1
890     END REPeat compare
900     LET come(p) = come(0)
910   END FOR item
920   FOR k = 1 TO num : send(k) = come(k)
930   send(0) = num
940 END DEFine
```

The following additional lines test the sort procedure.

```
10 REMark Test Sort
20 DIM row$(7,3),back$(7,3)
25 LET row$(0) = 7
30 FOR k = 1 TO rows$(0) : READ row$(k)
40 sort row$,back$
50 PRINT back$ (1 to back$(0))!
60 DATA "EEL", "DOG", "ANT", "GNU", "CAT", "BUG", "FOX"
```

Output: ANT BUG CAT DOG EEL FOX GNU

COMMENT

This program illustrates how easily you can handle arrays in TONTO BASIC. All you have to do is use the array name for passing them as parameters or for printing the whole array. Once the procedure is saved you can use **MERGE MDV1_sort** to add it to a program in main memory.

SCREEN PRESENTATION TECHNIQUES

You now have enough understanding of techniques and syntax to handle a more complex screen layout. Suppose you wish to represent the hands of four card players. A hand can be represented by something like:

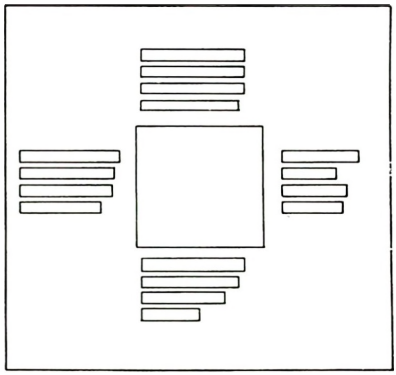
Card table layout

```
H: A 3 7 Q
C: 5 9 J
D: 6 10 K
S: 2 4 Q
```

To help the presentation the Hearts and Diamonds could be printed in white **INK** on light grey **PAPER**, and the Clubs and Spades in black **INK** on white **PAPER**. The general background could be black, and a table could be represented by a mixture of colours.

Method

Since a substantial amount of character printing is involved it is best to plan the layout in terms of a character grid. You can see that you need to provide for twelve lines of cards with some space between the cards, the table and the edge of the screen. A first guess at the layout is shown in the figure:



It is useful to recall that the default listing window is 20 lines of 80 characters. Down the screen the layout requires at least:

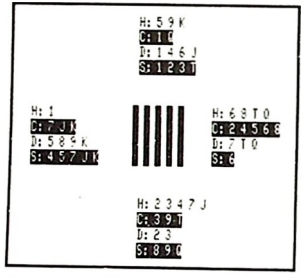
```
  8  =  two blocks of four lines for the cards
 +4  =  space above and below each block of cards
 +T  =  size of the table in lines
```

If the table is made four lines high there is then room for a one character border around the picture. This gives the following layout inside the border:

```
line 0          blank
     1  to 4    North hand
     5  to 6    blank
     7  to 10   Table, and West and East hands
     11 to 12   blank
     13 to 16   South hand
     17         blank
```

Experiment shows that the table appears square if it is 10 characters wide, and is centered if its left edge is at column 33.

The maximum length of a suit in any hand is 13. Allowing for a space before each card, and for the suit code (eg "H:"), the display of a hand is at most 28 characters wide. If the East hand starts at column 47 there is room for the widest hand. At the same time the gap between the table and the East hand is about the same as that between the table and the North hand. Similarly the West hand should start at column 2. However, this produces an asymmetrical effect in the display. If it is assumed that a run of more than 8 cards is unlikely then the West hand may be positioned at column 13 to give a more balanced display, with a slight risk of West's cards over-writing the table.



Window

Initially 480x200 at 0x0 - cleared to dark grey reduced to 468x180 at 6x10 (characters are 6x10 pixels) and cleared to black.

Table

10x4 characters with the top left-hand corner AT 7,33. Stripes produced by alternating light and dark grey.

Hands

Room for eight card symbols, longer runs will overwrite the table for West's hand only. Initial positions are:

```
NORTH  AT 1,34
EAST   AT 7,47
SOUTH  AT 13,34
WEST   AT 7,15
```

Character size

Default of 6 pixels wide, 10 pixels deep.

Colour

	Ink	Paper
Hearts	white	dark grey
Clubs	black	white
Diamonds	white	dark grey
Spades	black	white
background		black
border		dark grey
table		dark/light grey

Variables

card(52)	stored card numbers
sort(13)	used to sort each hand
tok$(4,2)	stores tokens H:, C:, D: and S:
ch_suit$	table to convert card value to A,1 to 9,T,J,Q,K
c	used as control variable when shuffling and splitting
ran	random position for card exchange
temp	holds card during exchange
item	card to be inserted in sort
dart	pointer to find position in sort
comp	hold card number in sort
lin	holds current line during table layout
col	holds current column within table layout
p	points to card position
seat	current hand being printed
ac	left edge of current hand
dn	current line in current hand
row	current suit in current hand
lin$	cards in current suit

max	maximum card number that could be part of the current suit
n	value of current card within suit
ch$	character representing value of current card

Procedures

shuffle	shuffles 52 cards
split	splits cards into four hands and calls *sortem* to sort each hand
sortem	sorts 13 cards into ascending order
layout	provides background colour, border and table
printem	prints each line of card symbols
getline	gets one row of cards, converts numbers into the symbols A,2,3,4,5,6,7,8,9,T,J,Q,K and prefixes with the suit token

Program outline

1. Declare arrays, pick up 'tokens' and place 52 numbers in array card
2. Shuffle cards
3. Split into 4 hands and sort each
4. OPEN a screen window, set a border and display the table
5. Print the four hands
6. CLOSE the screen window

Program

```
 10 DIM card(52),sort(13),tok$(4,2)
 15 LET ch_suit$="A23456789TJQK"
 17 RESTORE
 20 FOR k=1 TO 4 : READ tok$(k)
 30 FOR k = 1 TO 52 : LET card(k) = k
 40 shuffle
 50 split
 60 OPEN #6,scr_480x200a0x0
 70 layout
 80 printem
 90 CLOSE #6
100 STOP
200 DEFine PROCedure shuffle
210   FOR c=52 TO 3 STEP -1
220     LET ran = RND(1 TO c-1)
230     LET temp = card(c)
240     LET card(c) = card(ran)
```

```
250     LET card(ran) = temp
260   END FOR c
270 END DEFine
300 DEFine PROCedure split
310   FOR h = 1 TO 4
320     FOR c = 1 TO 13
330       LET sort(c) = card((h-1)*13+c)
340     END FOR c
350     sortem
360     FOR c = 1 TO 13
370       LET card ((h-1)*13+c) = sort(c)
380     END FOR c
390   END FOR h
400 END DEFine
500 DEFine PROCedure sortem
510   FOR item = 2 TO 13
520     LET dart = item
530     LET comp = sort(dart)
540     LET sort(0) = comp
550     REPeat compare
560       IF comp >=sort(dart-1) : EXIT compare
570       LET sort(dart) = sort(dart-1)
580       LET dart = dart-1
590     END REPeat compare
600     LET sort(dart)=comp
610   END FOR item
620 END DEFine
700 DEFine PROCedure layout
710   PAPER #6,4 : CLS #6
720   PAPER #6,0:WINDOW #6,468,180,6,10:CLS #6
730   FOR lin = 7 TO 10
731     AT #6,lin,33
732     FOR col = 1 TO 9 STEP 2
736       PAPER #6,4:PRINT #6," ";
738       PAPER #6,2:PRINT #6," ";
740     END FOR col
745   END FOR lin
750 END DEFine
800 DEFine PROCedure printem
820   LET p = 0
830   FOR seat = 1 TO 4
840     READ ac,dn
850     FOR row = 1 TO 4
860     getline
870     AT #6,dn,ac
880     PRINT #6,lin$
890     LET dn = dn + 1
900     END FOR row
910   END FOR seat
920 END DEFine
```

```
1000 DEFine PROCedure getline
1010   IF row MOD 2 = 0 THEN PAPER #6,6: INK #6,0
1020   IF row MOD 2 = 1 THEN PAPER #6,2: INK #6, 6
1030   LET lin$ = tok$(row)
1040   LET max = row*13
1050   REPeat one_suit
1060     LET p = p+1
1070     LET n =card(p)
1080     IF n>max THEN p = p-1:EXIT one_suit
1090     LET n = n MOD 13
1100     IF n=0 THEN n=13
1110     LET ch$=ch_suit$(n)
1120     LET lin$ = lin$&' '&ch$
1130     IF p = 52 : EXIT one_suit
1140     IF card(p) > card(p+1) : EXIT one_suit
1150   END REPeat one_suit
1160 END DEFine
1170 DATA "H:","C:","D:","S:"
1180 DATA 34,1,47,7,34,13,15,7
```

CONCLUSION

We have tried to show how you can use TONTO BASIC to solve problems. We have shown how simple tasks can be performed in simple ways. When the task is inherently complex, like manipulating arrays, TONTO BASIC enables it to be handled efficiently with maximum possible clarity.

If you were a beginner and you have worked through a fair proportion of this guide you have started well on the road to good programming. If you were experienced, we hope that you will appreciate and exploit the extra features offered by TONTO BASIC.

So enormous is the range of tasks which can be done with TONTO BASIC that we have only been able to touch a fraction of them in this guide. We cannot guess at which of the thousands of possibilities you will attempt, but we hope that you will find them fruitful, stimulating and fun. Happy programming!

Lists concepts in alphabetical order as follows:

Arrays
BASIC
Channels
Character set and keys
Clock
Coercion
Colour
Data entry
Data types, variables
Devices
Direct command
Error handling
Filetypes, files
Functions and procedures
Identifiers
Keywords
Maths functions
Memory
Microdrive
Operators
Permanent store
Pixel coordinates system
Program
Repetition
Screen
Segment
Slicing
Sound
Start up
Statement
String arrays, variables
String comparison
Windows

1 Summary

The following section describes concepts relating to TONTO BASIC and the TONTO hardware. It is best to think of the concept guide as a source of information: if there are any questions about BASIC or the TONTO itself which arise out of using the computer or the other sections of this manual, the concept guide may have the answer. Concepts are listed in alphabetical order using the most likely term for that concept. If you cannot find the subject, consult the index which tells you which page to turn to.

Where concepts are illustrated with examples, an example listed with line numbers is a complete program which can be entered and run. Examples listed without numbers are usually simple commands and it may not always be sensible to enter them into the computer in isolation.

ARRAYS

Arrays must be **DIM**ensioned (see page D2-32) before they are used. When an array is dimensioned, the value of each of its elements is set to zero, or a zero length string if it is a string array. An array dimension runs from zero up to the specified value. There is no limit on the number of dimensions which can be defined other than the total memory capacity of the computer. Array data is stored such that the last index defined cycles round most rapidly. In a **DIM** statement referring to a string array, the last index defines the maximum number of characters in each element.

Example

The array defined by

```
DIM array(4,2)
```

has elements stored as

For more details of arrays and array slicing, see sections 6 and 13 in Part B.

TONTO BASIC includes most of the functions, procedures and constructs found in many other dialects of BASIC. Some of these functions are superfluous in TONTO BASIC but are included for compatibility reasons:

GOTO	use **IF**, **REPeat**, etc
GOSUB	use **DEFine PROCedure**
ON GOTO	use **SELect**
ON GOSUB	use **SELect**

Some commands appear not to be present. You can always obtain them by using a more general function. For example, there are no **LPRINT** or **LLIST** statements in TONTO BASIC but you can direct output to a printer by opening the relevant channel and using **PRINT** or **LIST**.

LPRINT	use **PRINT#**
LLIST	use **LIST#** or **SAVE PRN**
VAL	not required in TONTO BASIC (see coercion, page C2-13)
STR$	not required in TONTO BASIC (see coercion, page C2-13)
IN	not applicable to this system
OUT	not applicable to this system

CHANNELS

A channel is a means by which data can be output to or input from a TONTO device. Before a channel can be used it must first be activated (or opened) with the **OPEN** command. Certain channels should always be active; these are the default channels, which allow simple communication with the TONTO via the keyboard and screen. When a channel is no longer of use it can be deactivated (closed) with the **CLOSE** command.

A channel is identified by a channel descriptor. This descriptor consists of # followed by the channel number. When the channel is opened, a device is linked to a given channel number and the channel is initialised. Thereafter, the channel is referred to only by its channel descriptor.

For example:

```
OPEN #5,PRN
```

links the printer to the channel number 5. When the channel is closed, only the channel descriptor should be specified.

For example:

```
CLOSE #5
```

Opening a channel requires that the device driver for that channel be activated. Usually there is more than one way in which the device driver can be activated. This extra information is appended to the device name and passed to the **OPEN** command as a parameter (see concept device). You can output data to a channel by **PRINT**ing to that channel. This is the same mechanism by which output appears on the TONTO screen. **PRINT** without a channel number outputs to its default channel.

For example:

```
10 OPEN #5, MDV1_test_file
20 PRINT #5, "this text is in file 'test_file'"
30 CLOSE #5
```

outputs the text

```
this text is in file 'test_file'
```

to the file *test_file*. It is important to close the file after all the accesses have been completed to ensure that all data is written.

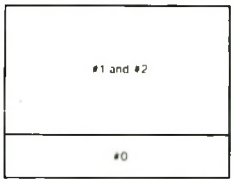
You can input data from a file in an analogous way using **INPUT**; data can be input from a channel one character at a time using **INKEY$**.

A channel can be opened as a console channel; output is directed to a specified window on the TONTO screen, and input is taken from the TONTO keyboard. When a console channel is opened, the size and shape of the initial window is specified.

The TONTO has three default channels which are opened automatically:

channel 0 - command and error channel
channel 1 - output channel
channel 2 - program listing channel

Each of these channels is linked to a window on the TONTO screen:



Channels 0 and 2 may not be closed.

The commands you can use associated with channels and their functions are summarised below:

OPEN	open a channel for I/O
CLOSE	close a previously opened channel
PRINT	output to a channel
INPUT	input from a channel
INKEY$	input a character from a channel
PAPER	select background colour for window channel
INK	select foreground colour for window channel
CLS	clear window channel to background colour

CHARACTER SET AND KEYS

In the following table, numberpad keys are indicated by a letter K in front of the numeric character.

Decimal	Hex	Keying	Display/Function
0	00	CTRL 0	null
1	01	CTRL A	soh
2	02	CTRL B	stx
3	03	CTRL C	etx
4	04	CTRL D	eot
5	05	CTRL E	enq
6	06	CTRL F	ack
7	07	CTRL G	bel
8	08	CTRL H	bs
9	09	TAB (CTRL I)	
10	0A	SHIFT RETURN (CTRL J)	New line/Command entry
11	0B	CTRL K	vt
12	0C	CTRL L	ff
13	0D	RETURN (CTRL M)	New line/Command entry
14	0E	CTRL N	so
15	0F	CTRL O	si
16	10	CTRL P	del
17	11	CTRL Q	dc1
18	12	CTRL R	dc2
19	13	CTRL S	dc3
20	14	CTRL T	dc4
21	15	CTRL U	nak
22	16	CTRL V	syn
23	17	CTRL W	etb
24	18	CTRL X	can
25	19	CTRL Y	em
26	1A	CTRL Z	sub
27	1B	ESC (SHIFT K*)	
28	1C	CTRL 1 or CTRL K1	
29	1D	CTRL 2 or CTRL K2	
30	1E	CTRL 3 or CTRL K3	
31	1F	CTRL 4 or CTRL K4	
32	20	Space	(Space)
33	21	SHIFT 1	!
34	22	SHIFT '	"
35	23	SHIFT 3	£

Codes to 20 hex are either control character or non printing characters. Alternative keyings are shown in brackets after the main keying.

Decimal	Hex	Keying	Display/Function
36	24	SHIFT 4	$
37	25	SHIFT 5	%
38	26	SHIFT 7	&
39	27	'	'
40	28	SHIFT 9	(
41	29	SHIFT 0	)
42	2A	SHIFT 8 or K*	*
43	2B	SHIFT =	+
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	30	0	0
49	31	1	1
50	32	2	2
51	33	3	3
52	34	4	4
53	35	5	5
54	36	6	6
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	SHIFT ;	:
59	3B	;	;
60	3C	SHIFT ,	<
61	3D	=	=
62	3E	SHIFT .	>
63	3F	SHIFT /	?
64	40	SHIFT 2	@
65	41	SHIFT A	A
66	42	SHIFT B	B
67	43	SHIFT C	C
68	44	SHIFT D	D
69	45	SHIFT E	E
70	46	SHIFT F	F
71	47	SHIFT G	G
72	48	SHIFT H	H
73	49	SHIFT I	I
74	4A	SHIFT J	J
75	4B	SHIFT K	K
76	4C	SHIFT L	L
77	4D	SHIFT M	M
78	4E	SHIFT N	N
79	4F	SHIFT O	O
80	50	SHIFT P	P

Decimal	Hex	Keying	Display/Function
81	51	SHIFT Q	Q
82	52	SHIFT R	R
83	53	SHIFT S	S
84	54	SHIFT T	T
85	55	SHIFT U	U
86	56	SHIFT V	V
87	57	SHIFT W	W
88	58	SHIFT X	X
89	59	SHIFT Y	Y
90	5A	SHIFT Z	Z
91	5B	ALT 9	[
92	5C	ALT 5	\
93	5D	ALT 0	]
94	5E	SHIFT 6	^
95	5F	SHIFT -	_
96	60	ALT 4	`
97	61	A	a
98	62	B	b
99	63	C	c
100	64	D	d
101	65	E	e
102	66	F	f
103	67	G	g
104	68	H	h
105	69	I	i
106	6A	J	j
107	6B	K	k
108	6C	L	l
109	6D	M	m
110	6E	N	n
111	6F	O	o
112	70	P	p
113	71	Q	q
114	72	R	r
115	73	S	s
116	74	T	t
117	75	U	u
118	76	V	v
119	77	W	w
120	78	X	x
121	79	Y	y
122	7A	Z	z
123	7B	ALT 7	{
124	7C	ALT 1	|
125	7D	ALT 8	}

Decimal	Hex	Keying	Display/Function
126	7E	ALT 6	~
127	7F	ALT -	
128	80	←	Left
129	81	ALT ←	
130	92	CTRL ←	
131	83	CTRL ALT ←	
132	84	SHIFT ←	
133	85	SHIFT ALT ←	
134	86	SHIFT CTRL ←	
135	87	SHIFT CTRL ALT ←	
136	88	→	Right
137	89	ALT →	
138	8A	CTRL →	
139	8B	CTRL ALT →	
140	8C	SHIFT →	
141	8D	SHIFT ALT →	
142	8E	SHIFT CTRL →	
143	8F	SHIFT CTRL ALT →	
144	90	↑	Up
145	91	ALT ↑	
146	92	CTRL ↑	
147	93	CTRL ALT ↑	
148	94	SHIFT ↑	
149	95	SHIFT ALT ↑	
150	96	SHIFT CTRL ↑	
151	97	SHIFT CTRL ALT ↑	
152	98	↓	Down
153	99	ALT ↓	
154	9A	CTRL ↓	
155	9B	CTRL ALT ↓	
156	9C	SHIFT ↓	
157	9D	SHIFT ALT ↓	
158	9E	SHIFT CTRL ↓	
159	9F	SHIFT CTRL ALT ↓	
160	A0		
161	A1	ALT A	
162	A2	ALT B	
163	A3	ALT C	
164	A4	ALT D	
165	A5	ALT E	
166	A6	ALT F	
167	A7	ALT G	
168	A8	ALT H	
169	A9	ALT I	
170	AA	ALT J	

Decimal	Hex	Keying	Display/Function
171	AB	ALT K	
172	AC	ALT L	
173	AD	ALT M	
174	AE	ALT N	
175	AF	ALT O	
176	B0	ALT P	
177	B1	ALT Q	
178	B2	ALT R	
179	B3	ALT S	
180	B4	ALT T	
181	B5	ALT U	
182	B6	ALT V	
183	B7	ALT W	
184	B8	ALT X	
185	B9	ALT Y	
186	BA	ALT Z	
187	BB	CTRL 5 or CTRL K5	
188	BC	CTRL 6 or CTRL K6	
189	BD	CTRL 7 or CTRL K7	
190	BE	CTRL 8 or CTRL K8	
191	BF	CTRL 9 or CTRL K9	
192	C0	SHIFT K7	f/Caps on
193	C1	SHIFT K7	—/Caps off
194	C2	DEL	½/Delete left one character
195	C3	SHIFT DEL	→/Insert
196	C4	CTRL DEL	↑/Remove left one character
197	C5	SHIFT TAB	↓/Back tab
198	C6	CTRL TAB	~/Format
199	C7	ALT RETURN	||/Alt return
200	C8		¾
201	C9		÷
202	CA	# or ALT 3	#
203	CB	ALT 2	@
204	CC		{
205	CD		} ©
206	CE		
207	CF		-
208	D0	START	Start
209	D1	RESUME	Resume
210	D2	REVIEW	Review
211	D3	SHIFT RECALL	List
212	D4	SHIFT REDIAL	Last 6
213	D5	SHIFT SPKR	Auto

Decimal	Hex	Keying	Display/Function
214	D6	SHIFT K6	Show
215	D7	SHIFT K9	Look
216	D8	SHIFT K#	Print
217	D9	SHIFT START	Blank
218	DA		
219	DB		
220	DC		
221	DD		
222	DE		
223	DF		
224	E0	RECALL	Recall
225	E1	SPKR	Speaker
226	E2	REDIAL	Redial
227	E3	SHIFT K1	Hold + shuttle
228	E4	SHIFT K2	Select
229	E5	SHIFT K3	End
230	E6	SHIFT K4	Dial
231	E7	SHIFT K5	Hold
232	E8	SHIFT K8	Time
233	E9		
234	EA		
235	EB		
236	EC		
237	ED		
238	EE		
239	EF		
240	F0	ALT K0	f0
241	F1	ALT K1	f1
242	F2	ALT K2	f2
243	F3	ALT K3	f3
244	F4	ALT K4	f4
245	F5	ALT K5	f5
246	F6	ALT K6	f6
247	F7	ALT K7	f7
248	F8	ALT K8	f8
249	F9	ALT K9	f9
250	FA	SHIFT F5	
251	FB	CTRL SHIFT F5	
252	FC		Special space
253	FD		Back tab (CTRL ignored)
254	FE		Special newline (CTRL ignored)
255	FF		

Note that codes 192-223 are control commands. Codes 192, 193, 208 to 218, 250 and 251 are not available to BASIC.

Codes 224-232 are used for telephony only and are not available to BASIC.

CLOCK

The TONTO contains a real time clock which continues to run on batteries when the computer is switched off.

The format used for the date and time is:

1983 JAN 01 12:09:10

COMMENT

Individual year, month, day, and time can all be obtained by slicing the string returned by the **DATE$** command.

The commands you can use associated with the clock and their functions are listed below:

SDATE	set the clock
ADATE	adjust the clock
DATE	return the date as a number
DATE$	return the date as a string
DAY$	return the day of the week

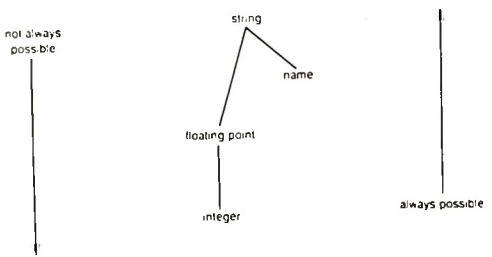
COERCION

If necessary, BASIC converts the type of unsuitable data to a type which allows the specified operation to proceed (see page B8-3).

The operators used determine the conversion required. For example, if an operation requires a string parameter but a numeric parameter is supplied, BASIC first converts the parameter to type string. It is not always possible to convert data to the required form, and if the data cannot be converted an error is reported.

The type of a function or procedure parameter can also be converted to the correct type. For example, the **LOAD** command requires a parameter of type name, but can accept a parameter of type string which is converted to the correct type by the procedure itself. Coercion of this form is always dependent on the way the function or procedure was implemented.

There is a natural ordering of data types on the TONTO. String is the most general type, since it can represent names, floating point and integer numbers. Floating point is not as general as string but it is more general than integer, since floating point data can represent integer data (almost exactly). The figure below shows this ordering diagramatically. Data can always be converted moving up the diagram, but conversion is not always possible moving down.



Examples

`a = b + c`	No conversion is necessary before performing the addition; conversion is not necessary before assigning the result to *a*
`a% = b + c`	No conversion is necessary before performing the addition but the result is converted to integer before assigning
`a$ = b$ + c$`	*b$* and *c$* are converted to floating point, if possible, before being added together. The result is converted to string before assigning
`LOAD "MDV1_data"`	The string *"MDV1_data"* is converted to type name by the **LOAD** procedure before it is used

COMMENT

You can write statements in TONTO BASIC which would generate errors in most other computer languages. In general, it is possible to mix data types in a very flexible manner.

For example...

```
PRINT "1" + 2 + "3"
LET a$ = '4'
LET a$ = 1 + a$ + '3'
PRINT a$
```

COLOUR

BASIC programs can display characters in four different colours (including black and white). These colours are black, red, qreen and white. On the TONTO monochrome monitor, colours are displayed as different intensities (shades of grey). The grey scale is four-level from black to white, from codes 0 and 1 black to codes 6 and 7 white.

Colours

The codes for colour selection are:

code	colour	shade
0	black	black
1	black	black
2	red	dark grey
3	red	dark grey
4	green	light grey
5	green	light grey
6	white	white
7	white	white

Examples

```
1    INK 0 : PAPER 0 : INPUT password$      black ink on black
                                            paper - invisible

2    PAPER 4 : CLS                          clear window to light
                                            grey
```

DATA ENTRY

The keyboard may be used to input:

- a line of BASIC
- data in response to an **INPUT** statement
- a single character in response to **INKEY$** or **PAUSE**

In the last case, the character input does not appear on the screen. In the first two cases the current input is reflected on the display as described below.

When a line of data is expected, a cursor appears in the window associated with the channel being used for input. Data characters are entered simply by typing the characters required on the keyboard. As each character is typed it is added to the current data-line and displayed on the screen at the current cursor position. The cursor is then moved one character position to the right - unless it is already at the right-hand edge of the window. In this case the cursor is moved to the left-hand edge of the next line down in the window, scrolling the contents of the window up one line if the cursor is already in the bottom line of the window.

The maximum number of characters that may be input is limited by the size of the window, such that the whole of the data-line may always be displayed. An attempt to input more characters than the window allows, or to input an unrecognised key-combination causes the generation of an error-tone. The offending character is not added to the data-line and the cursor is not moved.

Both the **RETURN** and **ENTER** keys terminate input and cause the data-line as displayed to be sent to the channel requesting input. If the input is in response to an **AUTO** or **EDIT** command, **ENTER** has the additional effect of terminating the **AUTO** or **EDIT** statement.

Key controls

Several keys are also usable as control or editing keys. These are described below:

A line of data or a line of BASIC program can occupy several lines on the screen. In the following explanation, such a multiple line of data or program is called a data-line, while a single physical line on the screen is simply called a line.

Cursor control keys

The cursor control keys are used to move the cursor both within a data-line and from one data-line to the next. Within a data-line the cursor moves one character position in the direction indicated by the arrow for each key depression. If the cursor is at the left hand edge of the window, ← left-arrow moves it to the right hand end of the line above. If the cursor is at the right hand edge of the window, → right-arrow moves it to the left hand end of the following line. If an attempt is made to move the cursor off the data-line using the left- or right-arrow key, the key will be ignored. ↑ up-arrow moves the cursor up one line, except if the cursor is at the top line of the display of the data-line. In this case it causes termination of input and the data-line is sent to the channel requesting input. If the input request is in response to part of an **EDIT** sequence, termination by up-arrow causes the previous line in the program-sequence to be displayed for editing. ↓ down-arrow has a similar effect, either moving down one screen line or, if the cursor is at the bottom of the data-line display -terminating input, sending the line and going onto the next program line in an edit-sequence.

The key combinations CTRL/left-arrow and CTRL/right-arrow may be used to skip to the start or end of the data-line respectively. At any position in the display, CTRL/up-arrow has the same effect as up-arrow used on the top line and CTRL/down-arrow has the same effect as down-arrow used on the bottom line.

RETURN

RETURN terminates the input and sends the data-line as displayed to the channel that was requesting input. Unlike **ENTER**, it does not cause termination of the **AUTO** or **EDIT** command, if any, that caused the request for input.

ALT/RETURN

ALT/RETURN has the same effect as **RETURN**, except that if the current cursor position does not coincide with the end of the data-line, all displayed characters of the data-line under and following the displayed cursor are redisplayed as spaces and removed from the data-line before it is sent to the channel.

DELete

Use of the **DEL** key changes the character immediately before the current cursor position in the data-line to a space and moves the cursor onto that space. The display is updated accordingly.

If the cursor is at the beginning of the data-line, **DEL** has no effect.

INS (SHIFT/DEL)

Use of the **INSERT** key combination enters a space at the current cursor position in the data-line, after first extending the data-line by one character and moving all characters under and to the right of the cursor by one position to the right. The display is updated accordingly. If the data-line is already equal to the maximum size defined by the window the data-line cannot be extended and the last character in the data-line is lost.

The cursor is not moved.

Remove (CTRL/DEL)

Remove is the inverse of **INS**. Use of the **REMOVE** key combination removes the character under the cursor from the data-line, moves all characters following by one position left, and shortens the data-line by one character. The display is updated to reflect the data-line, replacing the previously last character by a space. The cursor is not moved.

Break (CTRL/SPACE)

Use of the **break** key combination aborts any current processing and causes a prompt for the next command line.

DATA TYPES
VARIABLES

Integer

Integers are whole numbers in the range -32768 to +32767. Variables are assumed to be integers if the variable identifier is suffixed with a percent (%). There are no integer constants in TONTO BASIC; all constants are stored as floating point numbers.

Examples

1 counter%

2 size_limit%

3 this_is_an_integer_variable%

Floating point

Floating point numbers are in the range + (10-615 to 10615) with 8 significant digits. Floating point is the default data type. All constants are held in floating point form and can be entered using exponent notation.

Examples

1 current_accumulation

2 76.2356

3 354E25

4 25

String

A string is a sequence of allowable characters up to 32767 characters long. Variables are assumed to be type string if the variable name is suffixed by a $. String data is represented by enclosing the required selection of characters in either single or double quote marks. A string must be terminated with the same type of quote mark as it is started with. A string in double quote marks can be included inside a string enclosed in single quotes or vice versa.

Examples

1 string_variables$

2 "this is string data"

Name

Type name has the same form as a standard TONTO BASIC identifier and is used by the system to name devices.

Examples

```
1     MDV1_data_file

2     PRN
```

DEVICES

A device is a piece of equipment on the TONTO to which data can be sent and from which data can be received.

Since the system makes no assumptions about the ultimate I/O device which will be used, the I/O device can be easily changed and the data diverted between devices. For example, a program may want to output to a printer at some point during its run. If the printer is not available, the output can be diverted to a Microdrive file and stored. The file can be printed at a later date. I/O on the TONTO can be thought of as being written to and read from a **logical file** which is in a standard device independent form.

All device specific operations are performed by individual device drivers specially written for each device on the TONTO.

When a device is activated, a channel is opened and linked to the device. To open a channel device correctly, basic information must be supplied to initialise the device driver. This extra information is appended to the device name. The combination of a device name and appended information is called a **device specification**.

The **device specification** must conform to the rules for a BASIC type name though it is also possible to build it up as a string expression.

Each logical device on the system will attempt, where possible to provide defaults for omitted extra information.

Devices

The physical devices on the TONTO are referred to by the following names:

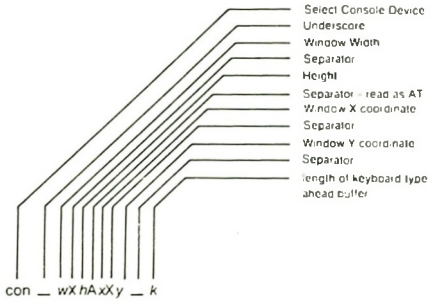
Name	Device
con	Console, consisting of keyboard and screen
mdv1	Microdrive 1, the left-hand microdrive
mdv2	Microdrive 2, the right-hand microdrive
prn	Printer, the optional printer
scr	Screen

When a channel is connected to a device, further information may be needed about the device. For the console and screen devices, this is the area of the screen to be used - the window specification. For the microdrives, this additional information is the name of the file to be used.

The next sections show how this information is expressed for each type of device.

Console

For the console device



con_wXhAxXy_k Console I/O

[wXh] - window width, height
[AxXy] - window X, Y coordinate
[k] - keyboard type ahead buffer length (bytes)

Default

con_480x200a0x0_128

Examples

```
1     OPEN #4,con_24x50a0x0_32

2     OPEN #8,con_24x50

3     OPEN #7,con_24x50a18x10
```

Screen

scr_wXhAxXy Screen Output

[wXh] - window width, height
[AxXy] - window X,Y coordinate

Default

scr_480x200a0a0x0

Examples

OPEN #4, scr_12x10a24x50

OPEN #5, scr_12x10

Microdrives

mdvn_*name* Microdrive File Access
[**mdv**] - select microdrive device
[n] - microdrive number
[*name*] - microdrive file name

Default

no default

Examples

1 OPEN #9, mdv1_data_file

2 DELETE mdv2_test_program

Printer

prn Printer output

Default

prn

Examples

1 OPEN #4, PRN

2 COPY mdv1_results TO prn

DIRECT COMMAND

BASIC makes a distinction between a statement typed in preceded by a line number and a statement typed in without a line number. Without a line number, the statement is a direct command and is processed immediately by the command interpreter. For example, if **RUN** is typed in on the command line and is processed, the effect is that the program starts to run. If a statement is typed in with a line number, the syntax of the line is checked and any detectable syntax errors are reported. A correct line is entered into the program and stored. A series of statements constitutes a program, which is only executed when the program is started with the **RUN** or **GOSUB GOTO** command.

Not all BASIC statements make sense when entered as a direct command. For example, **END FOR** and **END DEF**ine when entered by themselves in this way.

Not all BASIC statements make sense when included in a program. For example, if AUTO and EDIT are included in a program, they have no effect while the program is running.

Most BASIC statements can be used both as direct commands and within a stored program.

ERROR HANDLING Errors are reported by TONTO BASIC in a standard form:

At line *line_number error_text*

when the error is in a running program

error-text

when the error is in a direct command

Where the *line_number* is the number of the line where the error was detected and the *error_text* is one of the messages listed below:

- **Already exists**
  The file system has found an already existing file with the same name as a new file to be opened

- **Bad line**
  BASIC syntax error has occurred

  This error can also occur after an error in a function or procedure call. It prevents any lines being typed in which would modify the program. Typing **NEW** or **CLEAR** modifies this condition.

- **Bad medium**
  The medium is possibly faulty

- **Bad name**
  The file system has recognised the name, but there is a syntax or parameter value error.

  In TONTO BASIC it means that a name has been used out of context. For example, a variable has been used as a procedure

- **Bad parameter**
  There is an error in the parameter list of a system or BASIC procedure or function call.

  An attempt was made to read data from a write only device

- **Buffer full**
  An I/O operation to fetch a buffer full of characters filled the buffer before a record terminator was found

- **Catalogue in use**
  Can only occur when opening a file for write access or creating a new file. An activity has opened the catalogue file. Try again later

- **Channel not open**
  Attempt made to read, write or close a channel which has not been opened

- **End of file**
  End of file detected during input or **DATA** exhausted when **READ**ing

- **Error in expression**
  An error was detected while evaluating an expression

- **Internal tables full**
  The most likely cause of this error is that the program requires more memory, which cannot be allocated because the system's segment tables are full

- **In use**
  The file system has found that a file or device is already exclusively used

- **Media failure**
  A cartridge cannot be read

- **Not complete**
  An operation has been prematurely terminated

- **Not formatted**
  There has been an attempt to use a cartridge that is not formatted or not in place

- **Not found**
  File system, device, medium or file cannot be found

  BASIC cannot find an identifier. This can result from incorrectly nested structures

- **Not implemented**
  You have attempted to use a feature which is not implemented

- **Out of memory**
  BASIC has insufficient free store

- **Out of range**
  Usually results from an attempt to write outside a window or an attempt to reference an array element with an incorrect index

- **Out of sectors**
  Returned from microdrive access:
  Input: A sector of the file is missing
  Output: The cartridge is full

- **Overflow**
  Arithmetic overflow, division by zero, square root of a negative number, etc.

- **Parity error**
  Data read from a cartridge is corrupt

- **Read only**
  There has been an attempt to write data to a shared file

- **Sound queue full**
  Either the system sound queue or BASIC's sound queue is full

- **Write protected**
  The lug on the cartridge has been removed and the cartridge cannot be written to

- **Wrong format**
  The cartridge has been formatted, but not as a TONTO cartridge

Error recovery

After an error has occurred, the program can be restarted on the next statement by typing

CONTINUE

If the error condition can be corrected without changing the program, the program can be restarted on the statement which triggered the error. Type

RETRY

FILE TYPES AND FILES

All input/output on the TONTO is to or from a channel which references a logical device. Certain devices can support more than one channel at a time, e.g. a microdrive. Such a device is file-orientated.

Although all files contain data, each is expected to conform to one of the following types:

Text

Consists of strings interspersed with and terminated by a line feed character. Generated by **SAVE**, **OPEN_NEW** and **COPY**. Read using **INPUT**, **INKEY$**, **LRUN** etc.

Application

A special form of text file generated by the **PUBLISH** command. Read as an ordinary text file.

Image

Consists of an image of memory. Generated by the **SBYTES** command. Read using **LBYTES** or **INKEY$**.

FUNCTIONS AND PROCEDURES

BASIC functions and procedures are defined with the **DEFine FuNction** and **DEFine PROCedure** statements.

A function is activated (or called) by typing its name at the appropriate point in a BASIC expression. The function must be included in an expression because it is returning a value and the value must be used.

A procedure is activated (or called) by typing its name as the first item in a BASIC statement.

Data can be passed into a function or procedure by supplying a list of **actual parameters** when it is called. The **formal parameters** supplied with the definition are assigned the values of the actual parameters.

Since the actual parameters are actual expressions, they must have an actual type associated with them. The formal parameters are merely used to indicate how the actual parameters must be processed and so have no type associated with them. The items in each list of parameters are paired off in order when the function or procedure is called and the formal parameters become equivalent to the actual parameters. There are two distinct ways of using parameters:

- If the actual parameter is a single variable, then any data assigned to the formal parameter in the function or procedure is also assigned to the corresponding actual parameter

- If the actual parameter is an expression, assigning data to the corresponding formal parameter has no effect outside the procedure. Note that a variable can be turned into an expression by enclosing it within brackets

It is not always necessary to specify a full set of actual parameters. For example the BASIC **PRINT** statement is implemented as a procedure and can accept a variable number of parameters of varying types. Any formal parameter which has no corresponding actual parameter may have a spurious type and/or value. An error will usually result if the parameter is accessed.

Variables can be defined to be local to a function or procedure with the **LOCal** statement. Local variables have no effect on similarly named variables outside the function or procedure in which they are defined, and so allow greater freedom in choosing sensible variable names without risking corrupting external variables. Local variables are available to any nested function or procedure, unless they are again defined to be local.

Functions and procedures in BASIC can be used recursively. That is, a function or procedure can call itself either directly or indirectly.

IDENTIFIER

A BASIC identifier is a sequence of letters, digits and underscores.

Examples

1 a

2 limit_1

3 current_guess

4 BOND_007_james

An identifier must begin with a letter followed by a sequence of letters, digits and underscores, and can be up to 255 characters long. Upper and lower case characters are equivalent.

BASIC keywords cannot be used as identifiers. There are other reserved words in BASIC which cannot be used as identifiers. For a full list of reserved words see Appendix 2.

Identifiers are used in the BASIC system to identify variables, procedures, functions, repetition loops, etc.

WARNING

No meaning can be attributed to an identifier other than its ability to identify constructs to BASIC. BASIC cannot infer the intended use of an identifier from the identifier's name.

KEYWORDS

Keywords are identifiers which are defined in the TONTO BASIC Keyword Reference Guide. Keywords have the same form as a BASIC standard identifier. The case of the keyword is not significant. Keywords are echoed as a mixture of upper and lower case letters and are always reproduced in full. The upper case portion indicates the minimum you need type in for the keyword to be recognised by the system.

Already existing keywords cannot be used as ordinary identifiers within a BASIC program.

Certain further keywords are reserved in TONTO BASIC for possible future use. These are:

ARC
ARC_R
BAUD
BLOCK
BORDER
CIRCLE
CIRCLE_R
CURSOR
DIR
ELLIPSE
ELLIPSE_R
EXEC
EXEC_W
FILL
FLASH
FORMAT
KEYROW
LINE
LINE_R
MODE
MOVE
NET
OVER
PAN
PENDOWN
PENUP
POINT
POINT_R
RECOL
SCALE
SCROLL
SEXEC
STRIP
TURN
TURNTO

Use of these words in a BASIC program will result in an error report being returned to the screen.

There are other reserved words in BASIC which cannot be used as identifiers. For a full list of reserved words see Appendix 2.

MATHS FUNCTIONS TONTO BASIC has the standard trigonometrical and mathematical functions

Function	Name	Usual algebraic notation
COS	cosine	cos α
SIN	sine	sin α
TAN	tangent	tan α
COT	cotangent	cot α
ACOS	arccosine	cos-1 α
ACOT	arccotangent	cot-1 α
ASIN	arcsine	sin-1 α
ATAN	arctangent	tan-1 α
EXP	exponential	e^x
LN	natural logarithm	ln
LOG10	common logarithm	log
INT	integer	$\lfloor x \rfloor$
ABS	absolute value	$\lvert x \rvert$
RAD	convert to radians	
DEG	convert to degrees	
PI	value of π	π
SQRT	square root	$\sqrt{x}$
RND	random number	

MICRODRIVES

Microdrives provide the main permanent storage on the TONTO. Each Microdrive cartridge has a potential capacity of up to about 100 Kbytes.

Each cartridge must be formatted before use and holds approximately 200 sectors of 512 bytes per sector. Each Microdrive file is identified using a standard BASIC file name.

A cartridge can be write-protected by removing the small lug on the righthand side.

General care

Physically, each Microdrive cartridge contains a 200 inch loop of high quality video tape which can revolve completely in 7 1/2 seconds.

NEVER touch the tape with your fingers or insert anything into the cartridge.

NEVER turn the computer on or off with cartridges in place.

ALWAYS store cartridges in their sleeves when not in use.

ALWAYS insert or remove cartridges from the Microdrive slowly and carefully.

ALWAYS ensure that the cartridge is firmly installed before starting the Microdrive.

NEVER move the TONTO with cartridge installed - even if it is not spinning.

NEVER touch the cartridge while the drive is in operation.

You must format a new blank TONTO Microdrive cartridge to prepare it for use by a TONTO. Any information already on a cartridge is lost when it is formatted. Formatting a cartridge takes 30 to 40 seconds. For details of how to format cartridges see the Handbook.

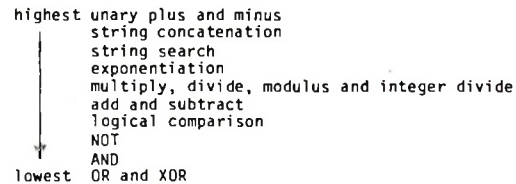
Operator	Type	Function
=	numeric	logical
	string	type 2 comparison
==	floating	almost equal **
	string	type 3 comparison
+	numeric	addition
-	numeric	subtraction
/	numeric	division
*	numeric	multiplication
<	numeric	less than
	string	type 2 comparison
>	numeric	greater than
	string	type 2 comparison
<=	numeric	less than or equal to
	string	type 2 comparison
>=	numeric	greater than or equal to
	string	type 2 comparison
<>	numeric	not equal
	string	type 3 comparison
&	string	concatenation
&&	bitwise	AND
||	bitwise	OR
^^	bitwise	XOR
~~	bitwise	NOT
OR	logical	OR
AND	logical	AND
XOR	logical	XOR
NOT	logical	NOT
MOD	integer	modulus
DIV	integer	divide
INSTR	string	type 1 string comparison
^	numeric	raise to the power
-	numeric	unary minus
+	numeric	unary plus

** almost equal is defined as equal within 1 part in 10^7

If the specified logical operation is true,a value not equal to zero is returned. If the operation is false, a value of zero is returned.

Precedence

The precedence of BASIC operators is defined below. If the order of evaluation in an expression cannot be deduced from this, the relevant operations are performed from left to right. The inbuilt precedence of BASIC operators can be overriden by enclosing the relevant sections of the expression in parentheses.



PERMANENT STORE

The TONTO contains 2K bytes of non-volatile memory known as the Permanent Store. Data stored in this memory is preserved when the TONTO is switched off, provided that battery power is available.

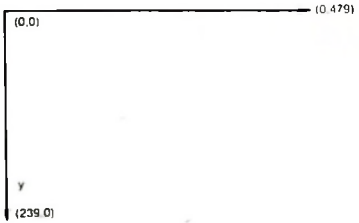
The Permanent Store is organised as a collection of entries identified by an entry number. Each entry may be from zero to 255 bytes in length and the identifying numbers are in the range 0 to 65535. All entries in the Permanent Store may be accessed from TONTO BASIC. However, entries with identifiers in the range 0 to 255 are used by the system. Altering the values of such entries may have undefined effects.

Entries in the Permanent store are manipulated using the following TONTO BASIC keywords:

SET_PSE	create or alter a Permanent Store Entry
PSE$	return the value of a Permanent Store Entry as a string of characters
DEL_PSE	delete a Permanent Store Entry

PIXEL COORDINATE SYSTEM

The pixel coordinate system is used to define the positions and sizes of windows on the TONTO screen. The coordinates system has its origin in the top left hand corner of the default window (or screen). The system uses the nearest character position. The pixel coordinate system is shown below:



PROGRAM

A program consists of a sequence of numbered lines, where each line may contain one or more TONTO BASIC statements. Line numbers are in the range of 1 to 32767.

Example

```
1     10 PRINT "This is a valid line number"

2     10 REM a small program
      20 PRINT "Guess a number between 1 and 100"
      30 PRINT "Double it and add 1"
      40 PRINT "Multiply by 3 and add 4"
      50 PRINT "Add 5 and divide by 2"
      60 INPUT "What number do you get to?"!ans
      70 LET orig=(ans-6) DIV 3
      80 CSIZE 3,1:CLS
      90 PRINT "Your number was"!orig
```

REPETITION

Repetition in TONTO BASIC is controlled by two basic program constructs. Each construct must be identified to TONTO BASIC.

```
REPEAT identifier                FOR identifier = range
   statements                       statements
END REPEAT identifier            END FOR identifier
```

These two constructs are used in conjunction with two other TONTO BASIC statements:

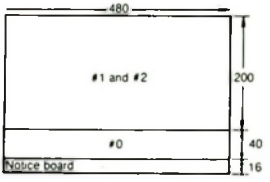
```
EXIT identifier                  NEXT identifier
```

Processing a **NEXT** statement either passes control to the statement following the appropriate **FOR** or **REPeat** statement or, if a **FOR** range has been exhausted, to the statement following the appropriate **END FOR**.

Processing an **EXIT** statement passes control to the statement after the **END FOR** or **END REPeat** selected by the **EXIT** statement. If an **EXIT** is used in a loop, the loop must be terminated by **END FOR** or **END REPeat**. **EXIT** can be used to exit through many levels of nested repeat structures. **EXIT** should normally be used in **REPeat** loops to terminate the loop on some condition.

SCREEN

The screen is 480 pixels across and is 256 pixels deep. The bottom 16 pixels are used to display the notice board and are not available to BASIC programs (see Handbook). The display comprises a four level scale from black to white.



When BASIC is entered, the total screen area is composed of 3 windows.

Window #0 is reserved for inputting BASIC commands and program lines, and displaying BASIC error messages. The window is initially defined to be the full width of the screen and occupies the last four lines above the Noticeboard.

Window #2 is reserved for displaying program lines as they are entered, or as they are requested by the LIST command to the default device. This window is defined to be the full width of the screen and occupies the remaining 20 lines above window #0.

Window #1 is the default application display and input window. It initially occupies exactly the same area as the listing window but has a background colour of dark grey.

The size and colour attributes of window #2 can, of course, be changed by a BASIC program. All window-based commands (e.g. **CLS, AT, PRINT**) are directed to this window if the channel parameter is omitted from the command.

SEGMENT

The memory on the TONTO is organised into segments whose size is a multiple of 512 bytes. Segments may be moved around in the memory by the system manager to make room for loading applications, creating new segments and so on. When accessing memory within a segment, it is necessary to fix the position of the segment for the duration of the access. Such action is termed "freezing the segment". When access to a segment has finished, the segment will remain frozen unless explicitly "thawed".

The user of BASIC can obtain use of extra memory with the **SEGMENT** function. BASIC will ensure that such a segment is frozen while it is accessed, and thawed afterwards, to make maximum use of the system's resources. Segments may be accessed from BASIC using **PEEK, PEEK_W, PEEK_L, POKE, POKE_W, POKE_L, CALL, LBYTES** and **SBYTES**.

SLICING

Under certain circumstances, it is possible to reference more than one element in an array - that is, slice the array. The array slice can be thought of as defining a sub array or a series of sub arrays to BASIC. Each slice can define a continuous sequence of elements belonging to a particular dimension of the original array. The term array in this context can include a numeric array, a string array or a simple string (an array of characters).

It is not necessary to specify an index for the full number of dimensions of an array. If a dimension is omitted, slices are added which select the full range of elements for that particular dimension - that is the slice (0 TO...). TONTO BASIC can only add slices to the end of the list of array indices.

An array slice or a string slice may be specified as the source of an assignment statement. A string slice may be specified as the destination of such an asignment.

Examples

1	**PRINT array (3, 1 TO 2)**	Prints array (3, 1) and array (3, 2)
2	**PRINT letters$ (4 TO)**	Prints the contents of letters$ from the fourth character to the end
3	**PRINT num (4 TO 5, 6 TO 7)**	Prints num (4, 6), num (4, 7) num (5, 6) and num (5, 7)

WARNING

Assigning data to a sliced string variable may not have the desired effect. Assignments made in this way do not update the length of the string and so it is possible that the system will not correctly process the assignment. The length of a string variable is only updated when an assignment is made to the whole string.

SOUND

Sound on the TONTO is generated by the system's second processor. A sound consists of a sequence of tones, each defined by a pitch and a duration, or a silence and a duration. A zero length silence at the end of a sequence causes the sequence to be repeated so that sounds of indefinite length may be generated.

The sound generator is also used for system functions such as telephony and may be used by other concurrent activities. Contention for the sound generator is handled by restricting any sound to a maximum of 2 seconds, if another sound is waiting to be output. Only a small number of sounds may be queued for output.

A sound is queued for output using the **BEEP** command. This command can also be used to cancel all sounds if used with no parameters. The function **BEEPING** may be used to check for sound output, and it returns the number of sounds currently being queued or output by the instance of BASIC invoking the function. It cannot be used to determine if any other activity is using the sound generator.

START UP

BASIC is entered by keying 7 from the Top Level Menu. If BASIC is not already in store, the message "Loading BASIC" appears against BASIC's menu entry and the system searches both microdrives for the program ICLBASIC, starting with the right-hand drive. If ICLBASIC is not on any cartridge on the microdrives, the message "NEEDS BASIC" appears against BASIC's menu entry in inverse video. If ICLBASIC is found it is loaded.

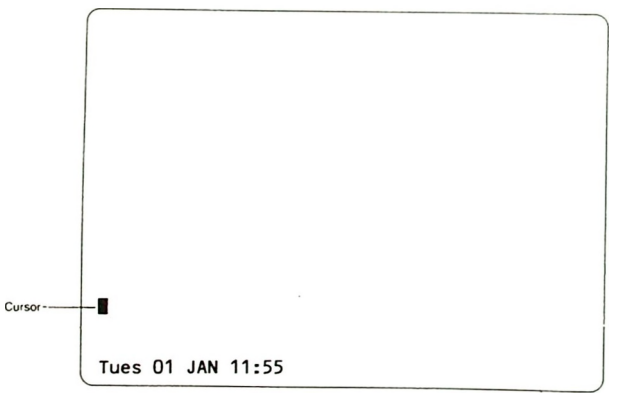
Once BASIC is in store it is activated by the system. The screen shows a copyright message for a few seconds and then clears. BASIC is ready for use when the cursor appears in the command window near the base of the screen.

It may happen that BASIC can be loaded but has insufficient store to complete its initialisation. In this case, the message "BASIC - NO STORE" appears in the noticeboard and the start up is abandoned.

BASIC may also be entered by selecting an application written in TONTO BASIC. The same start up sequence applies, except that any messages will appear against the menu entry for the selected application. When BASIC is ready to load the selected application, a command will appear in the command window, for example:

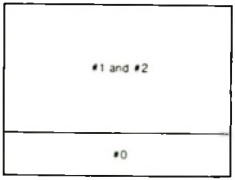
```
MRUN MDV1_CONFIGURATOR
```

and the command is executed to load and enter the application.



Default screen

The TONTO has three default channels which are linked to three default windows.



Channel 0 is used for listing commands and error messages, channel 1 for program output, and channel 2 for program listings. The default channel can be modified by omitting the optional channel specifier in the relevant command.

The initial channels are opened as though the following commands had been executed:

```
OPEN #0,CON 480 x 40a0x200 128
PAPER #0,0:INK#0,7:CLS#0
OPEN#1, CON 480 x200a0x0 128
PAPER#1,2 :INK#1,7:CLS#1
OPEN #2, CON 480x200a0x0 128
PAPER#2,1: INK#2,7:CLS#2
```

STATEMENT

A BASIC statement is an instruction to the TONTO to perform a specific operation.

For example...

```
LET a = 2
```

assigns the value 2 to the variable identified by a.

More than one statement can be written on a single line by spearating the individual statements from each other by a colon (:).

For example...

```
LET a = a + 2 : PRINT a
```

adds 2 to the value identified by the variable a and stores the result back in a. The answer is then printed out.

If a line is not preceded by a line number, the line is a direct command and BASIC processes the statement immediately. If the statement is preceded by a line number, the statement becomes part of a BASIC program and is added into the BASIC program area for later execution.

Certain BASIC statements can have an effect on the other statements over the rest of the logical line in which they appear, for example IF, FOR, REPeat, REMark.

STRING ARRAYS
STRING VARIABLES

String arrays and numeric arrays are essentially the same: however, there are slight differences in their treatment by BASIC. The last dimension of a string array defines the maximum length of the strings within the array. String variables can be any length. Both string arrays and string variables can be sliced (see page B11-2).

String lengths on either side of a string assignment need not be equal. If the sizes are not the same, either the right hand string is truncated to fit, or the length of the left hand string is reduced to match. If an assignment is made to a sliced string, the hole defined by the slice is padded with spaces, if necessary.

It is not necessary to specify the final dimension of a string array. Not specifying the dimension selects the whole string while specifying a single element picks out a single character, and specifying a slice defines a substring.

COMMENT

Unlike many BASICs, TONTO BASIC does not treat string arrays as fixed length strings. If the data stored in a string array is less than the maximum size of the string array then the length of the string is reduced.

WARNING

Assigning data to a sliced string array or string variable may not have the desired effect. Assignments made in this way do not update the length of the string and so it is possible that the system will not recognise the assignment. The length of a string array or a string variable is only updated when an assignment is made to the whole string.

STRING COMPARISON

The lexical ordering of characters for a string comparison is as below:

Order

```
space
f←½→↑↓½¼¾#@ 0(.)-0123456789AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp
OqRrSsTtUuVvWwXxYyZz !"£$%&'()*+,-/:;<=>?@[\] ^`{|}~
```

The relationship of one string to another may be:

equal All characters are the same or equivalent

lesser The first character of the first string which is different from the corresponding character in the second string, comes before it in the defined order

greater The first character of the first string which is different from the corresponding character in the second string, comes after it in the defined order

Types of comparison

Type	Character comparison	Embedded numeric sequences
0	case dependent	compared as characters
1	case independent	compared as characters
2	case dependent	compared as numbers
3	case independent	compared as numbers

Usage

type 0 Filename comparisons
type 1 Variable comparison
type 2 BASIC <,<=,=,>=,> and <>
type 3 BASIC = = (equivalence)

WINDOWS

Windows are areas of the screen which behave, in most respects, as though each individual window is a screen in its own right; that is, the contents of the window scroll when it has become filled by text, and a window can be cleared with the CLS command, and so on.

Windows can be specified and associated with a channel when that channel is opened. The current window size can be changed with the **WINDOW** command. Output can be directed to a window by printing to the relevant channel. Output is from the current cursor position, which can be positioned at any column and row within the window with the **AT** commnd.

Parts

The channel number of a window can be specified on an **INPUT** command. The cursor appears in the selected window to indicate that subsequent input will be to that window.

The **CLS** (clear screen) command accepts an optional parameter to define part of the window for its operation. This part is defined below:

parameter	part
0	all lines
1	all lines above the cursor
2	all lines below the cursor
3	the line containing the cursor
4	as 3, but only characters to the right of and including the cursor

The following commands are those you can use in association with windows:

Command	Function
AT	position cursor within a window
CLOSE	release a channel associated with a window
CLS	clear the screen
CSIZE	define character size for a window
INK	define the ink colour for a window
INPUT	receive data from a window
OPEN	associate a channel with a window and specify its initial size and position
PAPER	define the paper colour for a window
PRINT	display text in a window
UNDER	define underlining for a window
WINDOW	re-define a window

Part D Keywords Reference Guide

Lists TONTO BASIC keywords in alphabetical order as follows:

ABS
ACOS
ACOT
ADATE
AND
ASIN
AT
ATAN
AUTO
BEEP
BEEPING
BYE
CALL
CHR$
CLEAR
CLOSE
CLS
CODE
CONTINUE
COPY
COPY_N
COS
COT
CSIZE
DATA
DATE$
DAY$
DEFine
DEFine FuNction
DEFine PROCedure
DEG
DELETE
DIMN
DEL_PSE
DIM
DIV
DLINE
EDIT
ELSE
END
END DEFine
END FOR
END IF
END REPeat
END SELect
EOF
EXIT
EXP
FILL$
FOR
GOSUB
GOTO
IF
INCLUDE
INK
INKEY$
INPUT
INSTR
INT
LBYTES
LEN
LET
LIST
LN
LOAD
LOCAL
LOG10
LRUN
MERGE
MISTake
MOD
NEW
NEXT
NOT
ON
ON...GOSUB
ON...GOTO
OPEN
OPEN_IN
OPEN_NEW
OR
PAPER
PAUSE
PEEK
PEEK_W
PEEK_L
PI
POKE
POKE_W
POKE_L
PRINT
PSE$
PUBLISH
RAD
RANDOMISE
READ
REMAINDER
REMark
RENUM
REPeat
RESTORE
RETurn
RETRY
RND
RUN
SAVE
SBYTES
SDATES
SEGMENT
SELect ON
SET PSE
SIN
SQRT
STEP
STOP
TAN
THEN
TO
UNDER
VER$
WIDTH
WINDOW
XOR

1 Summary

Section 2 of the Keyword Reference Guide lists all BASIC keywords in alphabetical order. A brief explanation of the keyword's function is given followed by simple definition of its purpose in BASIC and examples of its use.

Each keyword entry shows to which group of operations it relates, under the heading, Type. For example ABS is a mathematical function and further information can be obtained from the Maths Function sections of the Concept Reference Guide.

Sometimes keywords are related to other keywords and it is necessary to deal with them together: for example **IF**, **ELSE**, **THEN**, **END IF**, are all listed under **IF**, as well as their separate entries. The notes section of each keyword tells you which keywords cannot be used in isolation (e.g. ELSE, **THEN**).

The full syntax of TONTO BASIC is listed in Appendix 1.

Type — Mathematical function

Purpose — This function returns the absolute value of a number. For zero or a positive number the value is not changed. For a negative number the value is equal to zero minus the number. The **ABS** function is often used to calculate the difference between two values when you do not know which is the larger of the two. For example **ABS**(4-5) or **ABS**(5-4) will always yield 1

Example — **100 Deviation = ABS (Last_Guess - New_Guess)**

Description — A mathematical function returning the unsigned value of its parameter

Format — **ABS** (<*numeric expression*>)

Type — Mathematical function

Purpose — To calculate an angle whose cosine is known

Examples — **PRINT ACOS (0.5)**

100 angle = DEG(ACOS(thi)) :REMark obtain angle in degrees

Description — A function returning the arc-cosine of its parameter. The result is in radian measure in the range 0 to π

Format — **ACOS** (<*numeric expression*>) range -1 to 1

Associated keywords — **ACOT, ASIN, ATAN, COS, COT, DEG, RAD, SIN, TAN**

Type	Mathematical function
Purpose	To calculate an angle whose cotangent is known
Example	**PRINT ACOT (0)**
Description	A function returning the arc-cotangent of its parameter. The result is in radian measure in the range $-\pi/2$ to $\pi/2$
Format	**ACOT** (<*numeric expression*>)
Associated keywords	**ACOS, ASIN, ATAN, COS, COT, DEG, RAD, SIN, TAN**

Type — Clock procedure

Purpose — This procedure allows the system clock to be adjusted forwards or backwards by a given number of seconds

Examples

```
ADATE -60     Set the clock back one minute

ADATE 30*60   Advance the clock by 30 minutes

120 b$=DATE$
130 IF b$(6 TO 11) = "OCT 28"
140   ADATE -60*60
150 END IF            :REMark BST ends
```

Description — A BASIC procedure that allows relative adjustments to be made to the system clock

Format — **ADATE** *<numeric expression>*

Associated keywords — **DATE, DATE$, DAY$, SDATE**

Type	BASIC operator
Purpose	To form the logical AND of two truth values
Examples	**60 IF hot AND sunny THEN go_to_the_beach** **If a >3 AND a <9 THEN PRINT "In range"**
Description	Thw two operands are treated as truth values, zero being false, non-zero true. The result is 1 (i.e. true) if both operands are true, 0 (i.e. false) if either or both of the oerands is false. To form the bitwise AND of two operands you should use the operator &&
Format	*<boolean expression>* **AND** *<boolean expression>*
Associated keywords	**NOT, OR, XOR**

Type	Mathematical function
Purpose	To calculate an angle whose sine is known
Examples	**PRINT ASIN (0.5)** **40 angle=DEG(ASIN(bval)) :REMark Angle from Sine**
Description	A function returning the arc-sine of its parameter. The result is in radian measure in the range $-\pi/2$ to $\pi/2$
Format	**ASIN (<***numeric expression***>)** range -1 to 1
Associated keywords	**ACOS, ACOT, ATAN, COS, COT, DEG, RAD, SIN, TAN**

Type — Screen handling procedure

Purpose — This procedure allows the cursor to be positioned at any character position in the current window prior to reading from or writing to the screen. It is a channel based command that defaults to the display channel. Both Row and Column arguments are defined as offsets in the current character height and width from the top left hand corner of the window

Examples

AT 0,0	Set the cursor to the top left hand of display window
30 **AT** #9,3,2	Set the cursor to row 3, column 2 in screen channel 9

Description — A channel based cursor control procedure which allows the cursor to be positioned relative to the selected window's character coordinates

Format — **AT** [#<*channel number*>,] <*row*>, <*column*>

Associated keywords — **CSIZE**, **PRINT**, **INPUT**, **INKEY$**, **CLS**

Notes — In other BASIC implementations this procedure may be called **POS** and/or may be an integral part of the **PRINT/INPUT** procedures

Type	Mathematical function
Purpose	To calculate an angle whose tangent is known
Examples	**PRINT ATAN (PI / 8)** **150 LET Erx = ABS (Angle - ATAN (SIN(Angle)/COS(Angle)))** **1000 a= ABS(b-ATAN(1))**
Description	A function returning the arc-tangent of its parameter. The result is in radian measure in the range $-\pi/2$ to $\pi/2$
Format	**ATAN** (<*numeric expression*>)
Associated keywords	**ACOS, ACOT, ASIN, SIN, COS, COT, DEG, RAD, TAN**

Type

BASIC command

Purpose

When generating a BASIC program it is common to assign to each line a number which increments uniformly. To save having to type in the line number each time, this command can be used to make the system present each new line number **AUTO**matically. Used on its own the system will present first 100, then 110, 120, 130 etc. Additionally both the first and subsequent line numbers may be controlled by specifying the starting value and the increment between lines. If the first parameter is omitted, it takes a default value of 100. If the second parameter is omitted, it takes a default value of 10. **AUTO** will continue until it is cancelled by the **BREAK** key or **SHIFT/ENTER** key sequence, or until the line number exceeds 32767.

Examples

AUTO	Present line numbers 100, 110, 120 etc
AUTO 5,5	Present line numbers 5, 10, 15, 20 etc
AUTO 10	Present line numbers 10, 20, 30 etc

Description

A command which allows you to enter lines without first typing in the number of the line

Format

AUTO [<*first line*>] [,<*increment*>]

Associated keywords

RENUM, EDIT

Notes

1 As **AUTO** is a direct command, it cannot reasonably be used in a program

2 Changing the line number (and then sending the line) causes an edited line to be inserted at the point specified in the program. The original line is left unchanged. This allows copying of lines but be careful. If the line number of the edited line corresponds to an existing line, that line is replaced

Type

Sound control procedure

Purpose

To permit or stop the generation of various sounds according to its parameters. Parameters must be supplied in pairs F,D where the pair has the following significance:

```
F = 1 to 255  :  tone frequency is 2400/F hertz
  = 0         :  no sound (i.e. silence)
D = 255       :  sound is to last indefinitely
  = 1 to 254  :  sound is to last for D/50 seconds
  = 0         :  the sound sequence is to be repeated from
                 the start
```

BEEP with no parameters stops any sound currently being output

Examples

```
120 BEEP 1,255  : REMark Continuous high pitch

180 IF BEEPING THEN BEEP   :REMark Stop the noise

320 BEEP 2,8,4,8,0,0       :REMark Siren

410 BEEP 8,8               :REMark Raspberry
```

Description

A BASIC procedure giving access to the system sound generator

Format

BEEP [*<frequency>*, *<duration>*,{*<frequency>*, *<duration>*]}

Associated keyword

BEEPING

Notes

1 As the TONTO tone generator is shared between many tasks, the program should restrict its sound output to a minimum

2 BEEP 2, 2 is not allowed

Type — Sound control function

Purpose — To discover whether all sounds generated by the program have completed

Examples — 210 IF BEEPING THEN PRINT "I'm a noisy program"

Description — A function returning a zero result if all BEEP commands have been fully processed; non-zero otherwise

Format — BEEPING

Associated keywords — BEEP

Type — Program control procedure

Purpose — To leave BASIC and release store to the system

Example — 200 IF play_again$ <> 'Y' THEN BYE_:REMark return to system menu

Description — A TONTO BASIC procedure that terminates the current instance of the BASIC interpreter

Format — BYE

Associated keyword — STOP

Type

System interface

Purpose

To commence execution of an assembler program from BASIC

Examples

```
CALL my_prog,32, 1,2,3,4
```

```
120 Prog=SEGMENT(4)                :REMark 2K required
130 LBYTES MDV1_myprog,Prog,0      :REMark Read in program
140 CALL Prog,2,parm1,parm2        :REMark Execute it
150 cc%=PEEK_W(Prog,0)             :REMark completion code
```

Description

A procedure providing an interface to the low level system facilities. The third and subsequent parameters are placed in the TONTO registers D1,....,D7,A0....,A5 in that order. A maximum of 13 additional parameters may be specified

Format

CALL *<segment identifier>,<offset>*{*,<register value>*}

Associated keywords

LBYTES, PEEK, POKE, SBYTES, SEGMENT

Note

Such an execution is monitored by BASIC. Do not attempt to use this facility unless you are a very experienced programmer

Type	Character translation function
Purpose	To translate an internal character code into the display symbol for that character
Examples	**PRINT CHR$(66)** Prints B **130 PRINT CHR$(194) :REMark print 1/2 as 1 char** **70 OPEN #4,prn** **80 PRINT #4,CHR$(15) :REMark Set condensed print**
Description	A function returning the internal character representation of its parameters
Format	**CHR$** (<*numeric expression*>) range 0 to 255
Associated keyword	**CODE**
Note	Not all values yield a displayable character

Type	System procedure
Purpose	To clear all variable store and release any store regained
Example	**CLEAR**
Description	A procedure which may reduce the store requirement of the BASIC program and sets all variables to an uninitialised state
Format	**CLEAR**
Associated keyword	**RESTORE**

Type — Device control procedure

Purpose — To close a channel and destroy the relationship between that channel and the associated device. If the channel is associated with a screen window then that window is deactivated.

Examples

```
CLOSE #9          Close channel identified by channel number 9

900 CLOSE #printer_channel  :REMark close printer_channel
```

Description — A procedure which causes a device or file to be closed and any end processing to be performed. It also releases associated store to the system

Format — **CLOSE** *#<channel number>*

Associated keywords — **OPEN, OPEN_IN, OPEN_NEW**

Note — Channels #0 and #2 are protected and cannot be cleared except by terminating BASIC

Type — Screen handling procedure

Purpose — To clear all or part of a screen window

Examples

```
10 AT 5,23:CLS4  :REMark clear previous input
10 CLS #4        :REMark Clear other window
```

Description — A channel based procedure which can clear all or part of a screen window to its current paper colour

Format — **CLS** [#*<channel number>*,][*<section>*]

where:
- *section* = 0 — whole window (default if no parameter)
- *section* = 1 — top excluding the cursor line
- *section* = 2 — bottom excluding the cursor line
- *section* = 3 — whole of the cursor line
- *section* = 4 — right end of cursor line including the cursor position

Associated keywords — **PAPER, WINDOW**

Note — Clearing the whole window sets the cursor position for that window to line 0, column 0. Clearing part of a window has no effect on the cursor position, except that any pending new lines is output first

Type — Character translation function

Purpose — To generate the internal code for the first character of a string

Examples

```
PRINT CODE ('BASIC')          Displays 66

70 FOR i=1 TO 80
80   rec$(i)=CHR$(CODE(rec$(i))*3/2+4  :REMark Encrypt
90 END FOR i
```

Description — A function returning the internal value of the first character of its parameter. It returns zero if the parameter is an empty string

Format — CODE (<*string value*>)

Associated keyword — **CHR$**

Type	Program control command
Purpose	To allow a program to be resumed at the next statement that it would have executed after halting due to a **STOP** command, an error or the **BREAK** key sequence
Examples	**CONTINUE**
Description	A procedure which allows an interrupted program to be resumed at the next logical statement provided that: - no program lines have been added, changed or deleted - no new variables have been created
Format	**CONTINUE**
Associated keywords	**RETRY**, **STOP**

Type — Data archival procedure

Purpose — To copy a data file to a device or another file. **COPY_N** removes the microdrive header from the file.

Examples

```
COPY MDV1_myprog TO MDV2_myprog

COPY_N MDV2_report1 TO PRN

360 COPY mdv1_datafile1 TO mdv2_databak1
```

Description — A command which allows data files or BASIC program files to be transferred between devices

Format — COPY[_N] *<file specification>* TO *<file specification>* | *<device specification>*

Associated keywords — **SAVE, PUBLISH, PRINT**

Type	Mathematical function
Purpose	To calculate the cosine of an angle
Example	**PRINT COS (RAD(45))**
Description	A function returning the cosine of its parameter. The parameter is in radian measure
Format	**COS** (<*angle in radians*>)
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COT, DEG, RAD, SIN, TAN**

Type	Mathematical function
Purpose	To calculate the cotangent of an angle
Example	**PRINT COT (PI/8)**
Description	A function returning the cotangent of its parameter. The parameter is in radian measure
Format	**COT** (<*angle in radians*>)
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COS, DEG, RAD, SIN, TAN**

Type — Screen handling procedure

Purpose — To change the character size for all subsequent output characters

Examples

```
280 CSIZE #5,3,1  :REMark Title in large characters

360 CSIZE 0,0  :REMark Revert to normal size
```

Description — A channel based procedure which alters the size attributes for subsequent output to the associated window. Characters may be 6 or 12 pixels wide, and 10 or 20 pixels high. A line across the screen holds 40 characters if they are 12 pixels wide, or 80 characters if they are 6 pixels wide. CSIZE has no effect if the channel is not a screen window

Format — CSIZE [#<*channel number*>,] <*width code*>,<*height code*>

where: <*width code*> is 0 or 1 for single width characters
2 or 3 for double width characters
: <*height code*> is 0 for single height characters
1 for double height characters

Associated keywords — **AT, PRINT, UNDER, WINDOW**

Note — The command fails *out of range* if a character of the new size will not fit within the window at the cursor position

Type	Intrinsic BASIC keyword
Purpose	To enable data values for **READ** statements to be embedded in a program
Example	**200 FOR a=1 TO 3** **210 READ area(a):READ town$(a)** **220 NEXT a** **900 DATA 7, "Liverpool",11,"Birmingham"** **910 DATA 32,"Norwich"**
Description	A keyword which must precede all lists of data in the program
Format	**DATA** *<expression>* { , *<expression>* }
Associated keywords	**READ, RESTORE**
Note	An implicit **RESTORE** is not performed before **RUN**ning a program

Type — Clock function

Purpose — To return the value of the TONTO clock as a floating point number

Examples

```
10 start=DATE
12 requested_characters = 'a'
20 REPEAT reactions
30   AT 0,0 : PRINT DATE-Start!'seconds'
40   IF inkey$=requested_character$ : EXIT reactions
50 END REPEAT reactions
```

Description — A function returning the value of the clock measured in seconds from the midnight preceding January 1st, 1970

Format — **DATE**

Associated keywords — **DATE$, DAY$**

Type

Clock function

Purpose

To return a textual representation of a time and date

Examples

```
PRINT DATE$          Prints the current TONTO date and time

PRINT DATE$(0)       Prints 1970 JAN 01 00:00:00

 80 now$=DATE$
 90 PRINT now$(13 TO 20)!"Finished"  :REMark time the message


190 b$=DATE$
200 PRINT #4,"Date:"!DAY$!b$(10 TO 11)!b$(6 TO 8)!b$(1 TO 4)
             :REMark date the report showing the day
```

Description

A function returning a date and time string representation of its parameter. If no parameter is supplied, the current TONTO date and time are given. The format of the string is:

YYYY MMM DD HH:MM:SS

where:

YYYY	is the year	1970, 1984 etc
MMM	is the month	JAN, FEB,... DEC
DD	is the day	01 to 31
HH	is the hour	00 to 23
MM	is the minute	00 to 59
SS	is the seconds	00 to 59

Format

DATE$ [(<*clock value*>)]

Associated keywords

DATE, DAY$

Type — Clock function

Purpose — To generate a textual representation of a day of the week from a value measured in seconds from the midnight preceding Jan 1st, 1970

Examples

```
140 IF DAY$ = 'FRI' THEN PRINT 'Yippeeee'

280 PRINT "Tomorrow will be"!DAY$(DATE + 24*60*60)
```

Description — A function returning a textual representation of its parameter, where the parameter is a valid clock value, and the result is the day of the week. If no parameter is given, **DAY$** returns the current day of the week

Format — **DAY$** [<*clock value*>]

Associated keywords — **DATE**, **DATE$**

Type	BASIC command structure
Purpose	To inform BASIC that a **PROCedure** or **FuNction** is about to be defined, or has just been defined
Description	A BASIC keyword which must precede declaration of a user function or procedure, or which must succeed **END** at the end of such a declaration
Associated keywords	**PROCedure, FuNction, END**
Note	This keyword never appears on its own

Type BASIC command structure

Purpose To define a user function

Examples

```
100 DEFine FuNction mean (a,b)
110   RETURN a + b/2
120 END DEFine mean


480 DEFine FuNction right$(string$,lnth)
490   RETurn string$(LEN(string$)-lnth+1 TO>
500 END DEFine right$

1000 DEFine FuNction getyesno$
1010   LOCal char$
1020   REPeat getchar
1030     INPUT "Enter Y or N"!char$
1040     IF char$(1) INSTR "NnYy" THEN EXIT getchar
1050   END REPeat getchar
1060   RETurn (CHR$(CODE(char$(1)) && 223)) :REMark uppercase
1070 END DEFine getyesno$
```

Description A BASIC construct which causes BASIC to define a user function

DEFine FuNction identifies the function and its parameter, if any. The sequence of statements between the DEFine function and the END DEFine constitute the function. The function definition may also include a list of formal parameters which supply data for the function. Both the formal and actual parameters must be enclosed in brackets. If the function requires no parameters, you do not specify an empty set of brackets.

Formal parameters take their type and characteristics from the corresponding actual parameters.

An answer is returned from a function by appending an expression to a RETurn statement. The type of the data is indicated by the character appended to the function name. A $ indicates string data, a % indicates integer data, no character indicates floating point data.

A function is activated by including its name at a suitable point in a BASIC expression.

Function calls in Basic can be recursive, that is a function may call itself directly or indirectly via a sequence of other calls.

Format

DEFine FuNction *<function name>* [(*<parameter>* {, *<parameter>* })]

Associated keywords

DEFine PROCedure, END DEFine, LOCal, RETurn

Note

Refer to pages B15-5 and C2-29 for more information.

Type BASIC command structure

Purpose To define a user procedure

Example

```
300 check inval,1,10
320 DEFine PROCedure check (value, min, max)
330   IF value < min OR value > max THEN
340     PRINT 'range check error'
350     STOP
360   END IF
370 END DEFine PROCedure check
```

Description A BASIC construct which causes BASIC to define a user procedure. **DEFine PROCedure** identifies the procedure and its parameters, if any. The sequence of statements between the **DEFine PROCedure** statement and the **END DEFine** statement constitutes the procedure. The procedure definition may also include a list of formal parameters which supply data for the procedure. The formal parameters must be enclosed in brackets for the procedure definition, but the brackets are not necessary when the procedure is called. If the procedure requires no parameters, you do not include an empty set of brackets in the procedure definition.

Formal parameters take their type and characteristics from the corresponding actual parameters.

Variables may be defined to be **LOCAL** to a procedure, local variables have no effect on similarly named variables outside the procedure. If required, local arrays should be dimensioned within the **LOCAL** statement.

You call a procedure by entering its name as the first item in a BASIC statement together with a list of actual parameters. Procedure calls in BASIC can be recursive, that is a procedure may call itself directly or indirectly via a sequence of other calls.

It is possible to regard a procedure definition as a command definition in BASIC; many of the system commands are themselves defined as procedures.

Format

DEFine PROCedure *<procedure name>* [(*<parameter>* { , *<parameter>* })]

Associated keywords

DEFine FuNction, END DEFine, LOCal, RETurn

Notes

1 Refer to section B7, B15-1 to B15-5 and C2-29 for further information on user defined procedures

2 The function or procedure name is not checked against the name of the function or procedure

Type	Mathematical function
Purpose	To calculate an angle expressed in degrees from an angle expressed in radians
Example	**PRINT DEG (PI/2)**
Description	A function returning the value of its parameter multiplied by 180, and divided by π
Format	**DEG** (*<radian angle>*)
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COS, COT, RAD, SIN, TAN**

Type	Microdrive command
Purpose	To delete a file
Examples	900 **DELETE MDV1_working_file** **DELETE MDV1_testprogram**
Description	A command allowing the deletion of a file from a microdrive cartridge
Format	**DELETE** *<file specification>*
Associated keywords	**SAVE, PUBLISH, OPEN_NEW**
Note	If the file does not exist, the error *not found* is reported

Type	Permanent store procedure
Purpose	To delete an entry in the permanent store
Examples	**DEL_PSE 400** **40 DEL_PSE my_bank_balance**
Description	A procedure whch releases the specified entry in the permanent store
Format	**DEL_PSE** *<numeric expression>*
Associated keywords	**PSE$, SET_PSE**
Notes	1 Do not delete entries 0 to 255. These are reserved for system use 2 It is not an error to delete an entry which does not exist

Type — BASIC keyword

Purpose — To define an array of a specific size to BASIC, so that BASIC can allocate the required store

Examples

```
10 DIM car_type$ (100,20), car_regs$ (100,9)

30 DIM revenue (12)
```

Description — A statement which dimensions arrays. Arrays must be declared before use

Format — **DIM** { ,*variable*> (<*maximum index*> { ,<*maximum index*> }) }

Associated keywords — **DIMN**, **LEN**

Notes

1 The **DIM** statement initialises all elements of an array

Numeric arrays: All elements set to 0
String arrays : All elements set to ""

2 The number of elements in any dimension is one more than the specified maximum index, since there is always an element with an index of 0

3 For a string array the last index is rounded up to a multiple of two and represents the maximum length of each string element. Element zero of the last dimension is, in fact, the current length of the string represented by that dimension and should not be altered explicitly

Type

BASIC function

Purpose

To return the maximum index of an array

Examples

```
230 FOR I = 1 TO DIMN(A$) : PRINT A$(I)

120 IF DIMN (x,3)<8 THEN STOP

100 DIM A (2,4,3)
110 PRINT DIM (A)     :REMark prints 2
120 PRINT DIM (A,1)   :REMark prints 2
130 PRINT DIM (A,2)   :REMark prints 4
140 PRINT DIM (A,3)   :REMark prints 3
150 PRINT DIM (A,4)   :REMark prints 0
```

Description

A function giving the maximum index of the dimension which is its parameter. The first parameter is the name of an array. The second (and subsequent) parameter is the number of the dimension. If the second parameter is omitted, a default of 1 is used, giving the size of the first dimension. If the second argument exceeds the number of dimensions, **DIMN** returns 0

Format

DIMN (<*array*> [,<*dimension number*>])

Associated keywords

DIM, **LEN**

Type	BASIC operator
Purpose	To indicate that an integer divide is required
Examples	**30 PRINT 'We each get'!apples DIV people! 'apples'**
Description	A binary operator performing integer division between its operands. The operands are converted to integer before division takes place. *a* DIV *b* produces the largest integer not exceeding a/b
Format	*<numeric expression>* **DIV** *<numeric expression>*
Associated keywords	**MOD**

Type	BASIC command
Purpose	To delete one or more lines from a BASIC program
Examples	**DLINE 100 to 170** **DLINE 10, 40, 90 TO 120, 180 TO 200**
Description	A command which removes lines from a BASIC program
Format	**DLINE** *<line range>* { , *<line range>* }

Type

BASIC command

Purpose

To request BASIC to enter editing mode. In this mode it is possible to make changes to a program one line at a time

Examples

```
EDIT

EDIT 200,10
```

Description

A command which causes BASIC to present previously entered lines for possible amendment

The **EDIT** command enters the BASIC line editor.

The **EDIT** command is closely related to the **AUTO** command; the only difference is in their defaults. **EDIT** defaults to a line increment of zero and thus edits a single line unless a second parameter is specified to define a line increment.

If the specified line already exists, the line is displayed and you can start editing. If the line does not exist, the line number is displayed and you can enter the line.

You can manipulate the cursor within the edit line using the standard TONTO keystrokes. When the line is correct, pressing ↵ enters the line into the program.

If an increment is specified, the next line in the sequence is edited, otherwise **EDIT** terminates.

If no line number is given, a default of 100 is used.

Format

EDIT [<*line number*>][,<*line increment*>]

Associated keywords

AUTO

Notes

1 Refer to pages B2-9 and C2-16 for an explanation of the editing keys

2 Changing the line number (and then sending the line) causes the edited line to be inserted at the point specified in the program. The original line is left unchanged. This allows copying of lines, but be careful. If the line number of the edited line corresponds to an existing line, that line is replaced

Type — BASIC command structure

Purpose — To separate the statements of an IF clause that are executed when the condition of the IF clause is true, from those that are executed when the condition is false

Examples

```
150 ELSE

10 IF Windy THEN sail : ELSE glide
```

Description — Part of the IF...THEN...ELSE structure

Format — [ELSE { <*statement*> }]

Associated keywords — **END IF, IF, THEN**

END

Type — BASIC command structure

Purpose — To inform BASIC that the end of a structure has been reached. **END** is always followed by the keyword which introduced the structure

Examples

```
380 END DEFine

870 END FOR I
```

Description — A control marker indicating the end of a structure

Associated keywords — **DEFine, FOR, IF, REPeat, SELect**

Note — This keyword never appears on its own

Type	BASIC command structure
Purpose	To indicate that the end of a user defined function or procedure has been reached
Examples	**END DEFine check** **END DEFine min** **END DEFine**
Description	A command indicating that the end of a routine has been found. If executed, this statement has the same effect as **RETurn**
Format	**END DEFine** [<*function name* or *procedure name*>]
Associated keywords	**DEFine FuNction, DEFine PROCedure, RETurn**
Notes	1 Refer to pages D2-25 to D2-27 and C2-29 for detailed explanation of user defined functions and procedures and see section B7 and pages B15-1 to B15-5 for details of how to use procedures and functions 2 The function or procedure name is not checked against the name of the function or procedure

Type	BASIC command structure
Purpose	To indicate that the end of a **FOR** loop has been found
Example	**END FOR count**
Description	A statement marking the end of a **FOR** loop. If executed, this statement will pass control to the statement after the corresponding **FOR** statement or, if the loop count has expired, will pass control to the next statement in sequence
Format	**END FOR** *<loop identifier>*
Associated keywords	**EXIT, FOR, NEXT**

END IF

Type	BASIC command structure
Purpose	To mark the end of an **IF** construct
Example	230 **END IF**
Description	A control statement delimiting the final clause of a conditional statement
Usage	**END IF**
Associated keywords	**ELSE, IF, THEN**

Type	BASIC command structure
Purpose	To indicate that the end of a **REPeat** has been found
Example	**120 END REPeat search**
Description	A statement marking the end of a **REPeat** loop. When executed, this statement passes control to the statement after the corresponding **REPeat** statement.
Format	**END REPeat** <*loop identifier*>
Associated keywords	**EXIT, NEXT, REPeat**

END SELect

Type	BASIC command structure
Purpose	To indicate the end of a **SELect...END SELect** clause
Example	**220 END SELect**
Description	A statement marking the end of a **SELect** clause
Format	**END SELect**
Associated keywords	**REMAINDER, SELect**

Type — Channel status function

Purpose — To return a logical value indicating whether there is further data which may be input from a channel or **READ** from **DATA**

Examples

```
540 IF EOF(#file) THEN EXIT display

2000 IF EOF(#4) THEN CLOSE #4
```

Description — A function used to determine whether the end of file has yet been reached on the specified channel

Usage — **EOF** (*#<channel number>*)

Associated keywords — **DATA, READ, RESTORE, INPUT, INKEY$**

Notes

1 Certain device types always return the same value:

SCR and PRN - EOF is always true

2 **EOF** with no parameter returns true if all **DATA** statements have been read

Type	BASIC control statement
Purpose	To allow exit from a **FOR** or **REPeat** loop
Example	**40 EXIT loop**
Description	A statement which transfers control to the statement after the related **END FOR** or **END REPeat** statement
Format	**EXIT** *<loop identifier>*
Associated keywords	**END FOR, END REPeat, FOR, NEXT, REPeat**

Type	Mathematical function
Purpose	To calculate the value of *e* to a given power
Examples	**PRINT EXP(2.4)** **100 LET y = 100*SIN(x) * EXP(x/30)**
Description	A function returning *e* to the power of the parameter
Format	**EXP** (*<numeric expression>*)
Associated keywords	**LN**

Type	String function
Purpose	To generate a string of specified length filled with one character, or a pair of characters
Examples	**PRINT FILL$ ('*', 80)** **30 var$ = FILL$ (' A', 100)**
Description	A function returning a string result of a given repetition of characters
Format	**FILL$ (**<*one or two character string*> **,** <*repeat count*>**)**

Type BASIC command structure

Purpose To allow a group of BASIC statements to be repeated a controlled number of times

Examples

```
FOR count = 1 to 40 STEP 2, 70 to 90 : PRINT count

10 FOR q = 1 to 10
20      IF NOT empty(q) THEN service q
30 END FOR q
```

Description A statement initialising a **FOR...END FOR** loop. The **FOR** statement allows a group of BASIC statements to be repeated a controlled number of times. You can use the **FOR** statement in both a long and a short form:

Short

The **FOR** statement is followed on the same logical line by a sequence of BASIC statements. The sequence of statements is then repeatedly executed under the control of the **FOR** statement. When the **FOR** statement is exhausted, processing continues on the next line. The **FOR** statement does not require its terminating **NEXT** or **END FOR**. Single line **FOR** loops must not be nested.

Example

```
350 FOR i=1 TO 300:READ element(i)
```

Long

The **FOR** statement is the last statement on the line. Subsequent lines contain a series of BASIC statements terminated by an **END FOR** statement. The statements enclosed between the **FOR** statement and the **END FOR** are processed under the control of the **FOR** statement

Examples

```
10 INPUT "data please" ! x
 20 LET factorial = 1
 30 FOR value = x TO 1 STEP -1
 40   LET factorial = factorial * value
 50   PRINT x !!!! factorial
 60   IF factorial>1E20 THEN
 70     PRINT "Very large number"
 80     EXIT factorial
 90   END IF
100 END FOR value


110 FOR i=0 TO 149
120  AT 3+i DIV 10,8*(i MOD 10)
130  PRINT item(i)
140 END FOR i
```

WARNING

A string or integer variable must not be used to control a FOR loop.

Format — FOR <*loop variable*> = <*required values*>

Associated keywords — **END FOR, EXIT, NEXT**

Notes — Refer to pages B4-1 to B4-4 for a detailed explanation of program looping

Type	Program control command
Purpose	To transfer processing to a specified line number but allow a **RETurn** to be made to the next statement after the **GOSUB**
Examples	20 **GOSUB** 300 140 **GOSUB** 200+100*selection
Description	A statement used to call a section of program as a subroutine
Format	**GOSUB** <*line number*>
Associated keywords	**GOTO, RETurn, PROCedure**
Note	This keyword is included for compatibility. Procedures should be used in preference

Type — Program control command

Purpose — To enable lines to be executed in an order other than that imposed by their line numbering

Examples

```
GOTO 10

100 GOTO 200+100* selection
```

Description — A statement used to transfer control to a specified or calculated line number unconditionally

Format — **GOTO** *<line number>*

Associated keyword — **GOSUB**

Notes

1 When executed as a command, **GOTO** <line number> has the same effect as **RUN** <line number>

2 If the specified line number does not exist, control is passed to the next highest line number. This applies equally to negative line numbers. Thus **GOTO**-1 causes execution to continue from the start of the program

Type

BASIC command structure

Purpose

To evaluate a conditional expression and use the result to control the subsequent actions of BASIC

Examples

```
600 IF value < 0 THEN PRINT 'Positive numbers only!!'

220 IF action$ = 'D' THEN
230    DELETE filename$
240 ELSE
250    IF action$ = 'V' THEN
260       COPY filename$ TO SCR
270    ELSE
280       PRINT 'Unrecognised action value - try again'
290    END IF
300    tidy_up_files
310 END IF
```

Description

A statement forming part of the IF...THEN...ELSE...END IF structure. This statement is the only part of that structure which is always required

You can use the IF statement in three forms:

Short

The **THEN** keyword is followed on the same logical line by a sequence of BASIC statements. If ELSE is included in the short form of IF, the ELSE must be included on the same logical line (see example 4). These statements are executed if the expression contained in the IF statement is true (evaluates to be non zero). **THEN** may be replaced by a : see example 3.

Examples

```
1     IF a = 32 THEN PRINT "Limit reached":a=0

2     IF test_data > maximum THEN LET maximum = test_data

3     IF a THEN PRINT "a is not zero":a= -1

4     IF x < min THEN min = x: ELSE x = x + 1
```

Long 1

The **THEN** keyword is the last item on the logical line. A sequence of BASIC statements is written following the IF statement. The sequence is terminated by the **END IF** statement. The sequence of BASIC statements is executed if the expression contained in the IF statement is true (evaluates to non zero). **THEN** may be omitted.

Example

```
 10 LET limit = 10: error_count = 0
 20 FOR try = 1 TO 20
 30   INPUT "type in a number" ! number
 40   IF number > limit THEN
 50     LET error count = error_count + 1
 60     PRINT "Out of range; error count is" ! error_count
 70     IF error_count > 5 THEN
 80       PRINT "Too many errors"
 90       EXIT try
100     END IF
110   END IF
120 END FOR try
```

Long 2

The **THEN** keyword is the last entry on the logical line. A sequence of BASIC statements follows on subsequent lines, terminated by the **ELSE** keyword. IF the expression contained in the IF statement is true (evaluates to be non zero), this first sequence of BASIC statements is processed. After the **ELSE** keyword a second sequence of BASIC statements is entered, terminated by the **END IF** keyword. If the expression evaluated by the IF statement is zero, this second sequence of BASIC statements is processed. **THEN** may be omitted.

Example

```
 10 LET limit = 10
 20 INPUT "Type in a number " ! number
 30 IF number > limit THEN
 40   PRINT "Range error"
 50 ELSE
 60   PRINT "Inside limit"
 70 END IF

100 IF a/2=INT(a/2) THEN
110   IF b/2=INT(b/2)
120     PRINT "Both even"
130   ELSE
135    PRINT "a even, b odd"
140   END IF
150 ELSE
160   IF b/2=INT(b/2) THEN
170     PRINT "a odd, b even"
180   ELSE PRINT "Both odd"
190   END IF
200 END IF
```

COMMENT

In all three forms of the IF statement the **THEN** is optional. In the short form it may be replaced by a colon to distinguish the end of the IF and the start of the next statement. In the two long forms it can be removed completely.

Format — IF *<condition>* **THEN** *<statement>* { *<statement>* }

Associated keywords — **ELSE, END IF, THEN**

Type — System interface

Purpose — To add additional keywords to BASIC and/or to introduce new devices

Example — **INCLUDE 3, "TLINK_DRIVER"**

Description — The first parameter is a search key with the following significance:

0 Search loaded programs only
1 Search loaded programs and microdrive 1
2 Search loaded programs and microdrive 2
3 Search loaded programs and both microdrives

where program is used in the wider TONTO sense rather than as a BASIC program

The second parameter is the 12 character name of a TONTO program. This program contains the code to support the new facilities together with information to allow the BASIC interpreter to add the new keywords and/or devices to its internal tables. BASIC searches for the program as specified by the search key and, if necessary, loads the program into store. The contents of such a program are beyond the scope of this document

Format — **INCLUDE** *<search key>,<TONTO program name>*

Type	Screen handling procedure
Purpose	To change the ink colour for all subsequent characters output to the specified channel
Examples	**10 PAPER 7 : INK 0** **INK #0,4**
Description	A channel based procedure which changes the foreground colour attributes of the associated window
Format	**INK** [#<*channel number*>,]<*ink colour value*>
Associated keyword	**PAPER**
Note	refer to pages B12-3 to B12-4 and C2-15 for a description of the colours generated by various ink values

Type

Data retrieval function

Purpose

To inspect the selected channel and obtain any available character

Examples

```
REPeat copier : PRINT INKEY$(#7) : END REPeat copier

90 IF INKEY$(5) = "" THEN PRINT "Too slow"

260 PRINT "Hit any key when ready,<ESC> to abort"
270 IF CODE(INKEY$(-1)) = 27 THEN EXIT overwrite
```

Description

A function which waits up to a specified time for a key to be pressed. The function returns either a null string if no key is pressed, or the value of the key pressed. The parameter is the maximum wait time specified in 20ms periods; if -1 the wait is indefinite

Format

INKEY$ ([#<*channel number*>,]<*wait period*>)

INKEY$

Associated keywords

INPUT, **PAUSE**

Notes

1 Characters obtained via this function are not echoed to the screen

2 INKEY$ may be applied to a channel associated with a microdrive file. In this case the wait period is ignored and either the next character in the file is returned or *end of file* is reported as appropriate

3 For a console channel, INKEY$ returns the next available character from the buffer associated with that channel. If the buffer is empty, the keyboard is inspected and if any key is depressed, its value is returned. A single key depression lasts a long time in computer terms and it is possible to read the same key depression twice if consecutive calls of INKEY$ are very close together

Type

Data retrieval procedure

Purpose

To obtain data values from a specified channel instead of from embedded data statements

Examples

```
10 INPUT a, b, c$

200 INPUT "A" ! a ! "B" ! b! "C" ! c

300 prompts$ = "what is your name?" : INPUT(prompt$) ! name$
```

Description

A statement to input values from a specified channel. **INPUT** allows you to enter data into a BASIC program. For a console channel, a cursor appears to indicate that input is expected and you must terminate each item by pressing the **RETURN** key. For a microdrive channel, each item is terminated by a line feed character.

You will notice that the syntax of *<IO list>* is a list of expressions separated by commas, semicolons, etc. as the the **PRINT** statement. Where one of these expressions is a variable, a value for the variable is input from the specified or default channel. Where the expression is not a simple variable (e.g. x+2 as opposed to x), the value of the expression is output to the channel. Note that any identifier can be turned into an expression by enclosing it in quotes.

Format

INPUT [#*<channel number>*,] *<IO list>*

Associated keywords

INKEY$, PRINT, READ

Notes

1 When **INPUT** is used with a **CON**sole channel the maximum number of characters that can be input is limited by the window size and is (height*width) -1 where height and width are in numbers of characters. If the window is only one character in size, the only valid key combinations in response to the **INPUT** are those that terminate input

2 When inputting a number the system will read characters until it finds a character that is not part of the number. For example, the following shows a short program and some examples of its output in response to various input strings

```
10 CLS
20 FOR I=1 TO 4 : INPUT D%!:PRINT, D%
30 FOR I=1 TO 4 : INPUT R!:PRINT, R
```

Input	Output	Comment
2	2	integer input stops at first non-digit
2.7	2	
2e01	2	
2e	2	
2	2	floating-point input stops when a syntactically correct number has been found
2.7	2.7	
2e01	2e0	
2e	error	

Type

String operator

Purpose

To calculate the starting index of the occurrence of one string within another

Examples

```
PRINT substr$ INSTR search_string$

50 IF NOT (guess$ INSTR word$) THEN hang_player

300 IF answer$(1) INSTR "NnYy": EXIT yesno
```

Description

A string operator which searches the right hand operand for the first occurrence of the left hand operand. If found, the value returned is the starting index of the matched string slice in the right hand operand; if not found, 0 is returned. Both operands are converted to strings before searching.

Zero can be interpreted as false, that is the substring is not contained in the given string. A non zero value, the substring's position, can be interpreted as true, that is the substring is contained in the specified string

Format

<string exprcssion> **INSTR** *<string expression>*

Type — Mathematical function

Purpose — To return the value of any number rounded down to the next integer

Examples

```
PRINT INT(-3.2)

40 LET rounded = INT (value +0.5)
```

Description — A function returning its real parameter converted to the next lower integer

Format — **INT** (<*numeric expression*>)

Type — Memory access procedure

Purpose — To allow a *memory image* data file to be loaded into memory at the specified start address

Examples

```
LBYTES MDV1_SCREEN, 131072

65 seg = SEGMENT(10) : LBYTES program$, seg, 0
```

Description — A procedure which allows direct cartridge to memory transfer

Format — **LBYTES** <*file specification*>, <*memory address*> |
<*segment identifier*>, <*offset*>

Associated keywords — **CALL, PEEK, POKE, SBYTES, SEGMENT**

Type	Dimensioning function
Purpose	To calculate the number of characters in a string
Examples	**PRINT LEN (string$)** **120 IF LEN(word1$) <> LEN(word2$) : PRINT 'not the same'**
Description	A function returning the length of the string supplied as its parameter
Format	**LEN (**<*string expression*>**)**

Type	Intrinsic BASIC command
Purpose	To allow values of variables to be initialised or updated
Examples	**dogs = dogs + 2** **LET Erx = ABS(Angle - ATAN(SIN(Angle)/COS(Angle)))**
Description	LET is an optional introduction to an assignment statement
Format	LET <*variable*> = <*expression*>

Type	BASIC command
Purpose	To list selected parts of a program to a channel, which is defaulted to the listing window
Examples	**LIST 50 TO 180** **LIST 10, 20, 90 TO 130**
Description	A command that lists the current program
Format	**LIST** [#<*channel number*>,] { <*selected lines*> }

Type	Mathmatical function
Purpose	To calculate the natural logarithm of a number
Example	**PRINT LN(20)**
Description	A function returning the natural logarithm of its parameter. Inverse logarithms can be calculated using **EXP**
Format	**LN** (<*numeric expression*>)
Associated keywords	**EXP, LOG10**

Type — Program storage command

Purpose — To retrieve a program from a device

Example — **LOAD MDV2_user_program**

Decription — A command that deletes any current program and retrieves the requested program from the specified device.

The effect of **LOAD** is to make BASIC read commands from the specified file. Normally, such commands are numbered BASIC program lines, resulting in a new program being stored for execution. Any input line which is not numbered is executed immediately.

A numbered line which is syntactically incorrect is nevertheless added to the stored program but is altered so that executing the line will cause an error. For example, if the file being loaded contains the line

```
200 PRINT "HELLO"; PRINT name$ : REM missing colon
```

You will find on listing the program that it is stored as

```
200 MISTake PRINT "HELLO"; PRINT name$ : REM missing colon
```

and an attempt to execute this line gives the message

```
At line 200 bad line
```

Format — **LOAD** *<file specification>*

Associated keywords — **LRUN, MERGE, MRUN, NEW, PUBLISH, SAVE**

Type — Variable control

Purpose — To allow variables and arrays to be local to the procedure or function in which they occur; their use in this function or procedure in no way affects the value of the variable with the same name outside it

Example

```
560 LOCAL val,num%,item$

800 LOCal c(3,30)  :REMark DIMension & LOCalise c
```

Description — A statement which can only be used inside a procedure or function definition. **LOCal** saves the values of the external variables named and restores these values when the function or procedure is completed. Arrays can be defined to be local by implicitly dimensioning them within the **LOCal** statement as in the example above.

The **LOCal** statement must precede the first executable statement in the function or procedure in which it is used.

Format — **LOCal** <*variable*> { , <*variable*> }

Associated keywords — **DEFine, DEFine FuNction, DEFine PROCedure**

Type	Mathematical function
Purpose	To calculate the common logarithm of a number to base 10
Example	**PRINT LOG10(27)**
Description	A function returning the common logarithm to base 10 of its parameter. Inverse logarithms (anti-logarithms) can be calculated using Y = 10 ^ X
Format	**LOG10** (<*numeric expression*>)
Associated keywords	**EXP, LN**

TYpe	Program storage command
Purpose	To load and run a BASIC program from a device
Example	**LRUN MDV2_game**
Description	This command is exactly equivalent to using **LOAD** followed by **RUN**
Format	**LRUN** <*file specification*>
Associated keywords	**LOAD, RUN**

Type — Program storage command

Purpose — To merge two or more whole or part programs together. It is very useful in building up a library of commonly used procedures and functions

Example — **MERGE MDV1_procedures**

Description — A command which causes a new program to be loaded from the specified device without erasing any previous program from store. Commands and program lines are effectively taken from the specified file as if they had been typed in at the keyboard, hence any clashes between line numbers will result in replacement of the original line. As with other variants of the **LOAD** command, any input line which has incorrect syntax has the word **MISTake** inserted between the line number and the body of the line. Upon execution, such a line generates an error

Format — **MERGE** *<file specification>*

Associated keywords — **LOAD, LRUN, MRUN**

Type	Special keyword
Purpose	To flag bad lines which have been loaded from a device
Example	10 **MISTake a-b+c**
Description	A keyword which causes an error on execution
Format	Only used by BASIC
Associated keywords	**LOAD, LRUN, MERGE, MRUN**

Type	BASIC operator
Purpose	To indicate that a modulus operation is required
Examples	**PRINT 7 MOD -3** Prints -2 **PRINT n MOD n** Prints 0 **PRINT -1 MOD 6** Prints 5
Description	A binary operator giving the modulus of its first operand in the base of the second. Both operands are converted to integer before the operation takes place.
Format	*<numeric expression>* **MOD** *<numeric expression>*
Associated keyword	**DIV**

Type — Program storage command

Purpose — To merge and run a BASIC program or programs

Example — **MRUN MDV1_new_data**

Description — This command is exactly equivalent to using **MERGE** followed by a **RUN** command

Format — **MRUN** <*file specification*>

Associated keywords — **LOAD, LRUN, MERGE**

Type — Direct command

Purpose — To clear out any program and variables which are held in the store to make way for a new program

Example — **NEW**

Description — A command which effectively deletes all traces of any previous program which was in BASIC's store

Format — **NEW**

Associated keyword — **CLEAR**

Type Program control command

Purpose To cause program execution to continue at the statement after its related **FOR** or **REPeat** statement, if the controlling variable is not exhausted

Examples

```
500 FOR i=1 TO 1000
505   IF EOF (#7) THEN EXIT i
510   INPUT #7,rec$          :REMark read stock rec
520   partnum(i)=rec$(1 TO 8)
530   stk_qty(i)=rec$(10 TO 16)
540 NEXT i
550 PRINT "Too many records"
560 STOP
570 END FOR i
```

Description A statement which delimits **FOR** loops in other BASICs. On the TONTO it has the same effect but may be placed anywhere within the body of the loop.

In a **FOR** loop for example, in conjunction with the **EXIT** statement, statements between the **NEXT** and **END FOR** statement will only be executed if the controlling variable expires. Premature loop termination as a result of obeying the **EXIT** statement, however, will inhibit the execution of this loop epilogue. This construct may be compared with **DO UNTIL** construct of other languages

Format **NEXT** *<loop identifier>*

Associated keywords **END FOR, END REPeat, FOR, REPeat**

Type BASIC operator

Purpose To complement a logical value

Example **IF NOT sunny THEN stay_indoors**

Description A unary logical operator which returns:

0 if the operand was non-zero
1 if the operand was zero

Format **NOT** *<boolean expression>*

Associated keywords **AND, OR, XOR**

Type	Program control command
Purpose	To allow multiway program control transfers. The value of the expression indexes the line number that is next executed. The value 1 selects the first line in the list, the value 2 the second line and so on
Examples	**110 ON X GOSUB 300, 400, 500** **380 ON Y GOTO 400, 500, 600**
Description	A statement providing multiple options in changing the order of execution of a program
Format	**ON** *<numeric expression>* **GOSUB** *<line number list>* **ON** *<numeric expression>* **GOTO** *<line number list>*
Associated keywords	**GOSUB, GOTO, SELect ON**
Note	These constructs are supplied only for compatibility. **SELect ON** should be used instead

Type — Device allocation procedures

Purpose — To associate a channel with a device or file

Examples

40 OPEN_NEW #8, PRN	open channel 8 to the printer
50 OPEN #4, SCR_ 60 x 60 a 60 x 60	open channel 4 to screen, creating a window size, 60 x 60 pixels at position 60,60
OPEN #5, f_name$	open file whose name is held in f_name$
OPEN_IN #9, "MDV1_file.exp"	open file MDV1_file.exp
OPEN_NEW #7, MDV1_data_file	open file MDV1_data_file
OPEN #6, CON_240x200a240x0_32	open channel 6 to the CONsole device creating a window occupying the right hand half of the default window with a 32 byte keyboard type ahead buffer

Description — A function which attributes to a file -

OPEN - for reading only
OPEN_IN - for reading only
OPEN_NEW - for writing only

If the channel is already associated with a device an implicit close is performed.

For all devices other than microdrive files **OPEN_IN** and **OPEN_NEW** are treated as equivalent to **OPEN** and give whatever access is available on the specified device i.e.

PRN - write only
SCR - write only
CON - read and write

If the specified device is a microdrive file **OPEN** and **OPEN_IN** are equivalent: the file must already exist and is opened for read access only. An individual file may be associated with several read channels at the same time. If **OPEN_NEW** is applied to an existing microdrive file it will report "already exists", otherwise a file of the specified name is created and opened for write access. Thus in TONTO BASIC, **OPEN** and **OPEN_IN** are equivalent for all devices.

Format — **OPEN #**<*channel number*>**,** <*device specification*>

Associated keywords — **CLOSE, PRINT, INPUT, INKEYS$**

Type — BASIC operator

Purpose — To form the logical inclusive OR of two truth values

Example

```
If windy OR wet THEN stay_at_home

60 IF val > max OR val < min:PRINT "Invalid"
```

Description — The two operands are treated as truth values, zero being false, non-zero true. The result is 1 (i.e. true) if either or both operands is true, 0 (i.e. false) if both operands are false. To form the bitwise OR of two operands you should use the operator ||

Format — <*boolean expression*> **OR** <*boolean expression*>

Associated keywords — **AND, NOT, XOR**

Type	Screen handling procedure
Purpose	To change the paper (background) colour for all subsequent characters written to the specified window channel
Example	10 **PAPER** 7 20 **OPEN #9, CON_60x60a120x30_32** 30 **PAPER #9, 3 : CLS #9**
Description	A procedure which changes the paper colour in the associated window
Format	**PAPER** [#<*channel number*>,]<*colour code*>
Associated keywords	**CLS, INK**
Note	Refer to pages B12-3 to B12-4 and C2-15 for the list of possible colours

Type	Timing command
Purpose	To cause a program to wait for a specified period of time (expressed in units of 20ms) or until a key is pressed
Examples	**PAUSE 50** wait 1 second or until a key is pressed **PAUSE 500** wait 10 seconds or until a key is pressed **200 PAUSE :REMark wait for a key depression**
Description	A procedure that executes as for **INKEY$** except that no value is ever returned. If used with no parameter, the pause is indefinite
Format	**PAUSE** [<*wait period*>]
Associated keyword	**INKEY$**

Type — Memory access functions

Purpose — To inspect the contents of memory at a specified address

Examples

```
PRINT PEEK_L (34578)

PRINT PEEK (seg3,4)

60 CALL srseg,2,parm1   :REMark start subroutine
70 cc%=PEEK_W(srseg,0)  :REMark get completion code
```

Description — Functions giving access to system memory. The amount of memory inspected is determined by the keyword used:

PEEK	returns 1 byte
PEEK_W	returns 2 bytes - address must be even
PEEK_L	returns 4 bytes - address must be even

Format — **PEEK** (*<memory address>*)

Associated keywords — **CALL, LYBYTES, POKE, POKE_W, POKE_L, SEGMENT, SBYTES**

Type — Mathematical function

Purpose — To calculate the value for π

Example — **PRINT PI**

Description — PI = 3.14159265

Format — **PI**

Type	Memory access procedure
Purpose	To change the contents of memory at the given address
Examples	**POKE_L 131072, -1** **70 POKE myseg, offset, value**
Description	Procedures which allow the contents of memory to be changed POKE returns 1 byte POKE_W returns 2 bytes POKE_L returns 4 bytes
Format	**POKE** *<memory address>,<expression>*
Associated keywords	**CALL, LBYTES, PEEK, SEGMENT, SBYTES**
Note	Poking of data into store is not recommended

Type — Data output procedure

Purpose — To transmit the required characters to a channel

Examples

```
PRINT #3, ATAN (PI / 8)

PRINT 'Hello' ! name$ ! 'How are you?'

30 OPEN #4,scr_240x190a240x0
40 PAPER #4,5
50 INK #4,0
60 AT #4,10,2:PRINT #4,"This is Basic version ";VER$
70 AT #4,12,0:PRINT #4,a,b\c
80 CLOSE #4
```

Description — A statement causing numeric and/or string values to be output to a channel.

Separators

! Best viewed as an intelligent space. Its normal action is to insert a space between items output on the screen. If the item will not fit on the current line, a line feed is generated. If the current print position is at the start of a line, a space is not output. ! affects the next item to be printed and therefore must be placed in front of the print item being printed. Also you must place a ; or a ! at the end of a print list if the spacing is to be continued over a series of **PRINT** statements

, BASIC tabulates output every 8 columns

\ Forces a new line

; Leaves the print head or cursor position immediately after the last item printed. Unless action is taken output is printed in one continuous stream

Format — **PRINT** [#<*channel number*>,] { *IO list* }

Associated keywords	**CLS, CSIZE, INPUT**

Type	Permanent store function
Purpose	To return the string value contained in a permanent store entry
Examples	**PRINT PSE$(283)** **60 PRINT "Program last run on"!PSE$(700)**
Description	A function returning the value of a permanent store entry
Format	**PSE$ (**<*numeric expression*>**)**
Associated keywords	**DEL_PSE, SET_PSE**

Type	Program storage command
Purpose	To save a BASIC program on microdrive cartridge in such manner as to allow the file to be loadable via the application selection menu
Example	**PUBLISH MDV2_super_prog**
Description	A special version of **SAVE** such that the program becomes eligible for display on the application selection menu
Format	**PUBLISH** *<file specification>*
Associated keywords	**SAVE, LOAD, MERGE, LRUN, MRUN**

Type	Mathematical function
Purpose	To calculate an angle expressed in radians from an angle expressed in degrees
Example	**PRINT RAD (45)**
Description	A function returning the value of its parameter multiplied by PI, and divided by 180
Format	**RAD** (*<angle in degrees>*)
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COS, COT, DEG, SIN, TAN**

Type	Mathematical function
Purpose	To allow the random number generator to be reseeded
Example	**40 RANDOMISE** :**REM**ark resets the seed for the random number generator **RANDOMISE 6** Sets the starting point for random number generator
Description	A function permitting the base for random numbers to be changed. If no parameter is supplied, various internal values are used
Format	**RANDOMISE** [*<numeric expression>*]
Associated keywords	**RND**
Note	If a constant is used as the parameter, the same sequence of numbers will be returned by successive invocations of the **RND** function

Type — Intrinsic BASIC command

Purpose — To enable variables to assume the value of data items in an embedded **DATA** statement

Examples

```
15 READ dx, dy, range, result

30 FOR side = 1 TO 2 : READ team$ (side), score (side)


70 DIM item(10),menu$(10,16)
80 FOR i=1 TO 10
90   READ item(i):READ menu$(i)
95 END FOR i
3000 DATA 1, "Create Cust",2,"Amend Cust"
3010 DATA 3, "Delete Cust",4,"Create Prod"
3020 DATA 5, "Amend Prod",6,"Delete Prod"
3030 DATA 7, "Enter Order",8,"Schedule Ord"
3040 DATA 9, "Enter Del",10,"Produce Invoice"
```

Description — A command which causes the next item of a data list to be copied into the variable(s) which follow

Usage — **READ** *<variable>* { , *<variable>* }

Associated keywords — **DATA, RESTORE**

Notes

1. An implicit RESTORE is not performed when the program is RUN
2. If variables are used as the operands of a data statement, they must be assigned before the **READ** statement is executed

Type — Pseudo constant

Purpose — To permit a **SELect** clause to cater for a response other than those explicitly defined

Examples

```
 90 SELECT ON var
100 ON var = 1 : PRINT "OK"
110 ON var = REMAINDER : PRINT 'Error'
```

Description — A pseudo constant which matches any value of a **SELect** variable, other than those values previously specified

Format — [**ON** *<select variable>*] = **REMAINDER**

Associated keywords — **END SELect**, **SELect**

REMark

Type — BASIC keyword

Purpose — To allow insertion of explanatory text into a program

Examples

```
10 REMark *** This is a comment ***
20 REMark everything is ignored...stop
```

Description — A keyword which permits the remainder of the line to be ignored

Format — **REMark** *<anything>*

Note — A **REMark** statement is delimited only by the end of a logical line. That is, a **REMark** statement must be the last of a line

Type

BASIC command

Purpose

To change the line numbers of a program such that all the statements stay in the same order, and such that all references are maintained

Examples

RENUM 100 to 200 ; 10,1

RENUM

Description

A command which renumbers the lines of a user's program and maintains most of the cross references within.

If there are expressed line numbers in the program

e.g. **120 RESTORE 300+30*severity**

the system attempts to renumber only the first part of the expression.

WARNING

You must never attempt to use **RENUM** to renumber program lines out of sequences, that is to move lines about the program.

Format

RENUM [<*first line*>][**TO** <*last line*>]
[;[<*new first line*>][,<*increment*>]]

Type BASIC command structure

Purpose To define the start of a **REPeat** loop

Examples

```
300 REPeat loop
```

Description A statement which is the start of a **REPeat...END REPeat** loop. **REPeat** can be used in both long and short forms:

Short

The **REPeat** keyword and loop identifier are followed on the same logical line by a colon and a sequence of BASIC statements. **EXIT** resumes normal processing at the next logical line.

Example

```
REPeat wait: IF inkey$ < >"" THEN EXIT wait
```

Long

The **REPeat** keyword and the loop identifier are the only statements on the logical line. Subsequent lines contain a series of BASIC statements terminated by an **END REPeat** statement.

The statements between the **REPeat** and the **END REPeat** are repeatedly processed by BASIC.

Examples

```
 10 LET number = RND(1 TO 50)
 20 REPeat guess
 30   INPUT "What is your guess?", guess
 40   IF guess = number THEN
 50    PRINT "You have guessed correctly"
 60    EXIT guess
 70   ELSE
 80    PRINT "You have guessed incorrectly"
 90   END IF
100 END REPeat guess
```

```
10 OPEN_IN #6,MDV1_tranfile
20 REPeat readloop
30   IF EOF (#6) THEN EXIT readloop
40   INPUT #6,rec$
50   process record
60 END REPeat readloop
70 CLOSE #6
```

COMMENT

Normally at least one statement in a **REPeat** loop is an **EXIT** statement.

Format — **REPeat** <*loop identifier*>

Associated keywords — **END**, **EXIT**, **FOR**, **NEXT**

Type	BASIC command
Purpose	To set the data pointer to a given line so that subsequent **READ** statements will access data from that point
Examples	**10 CLS : RESTORE :REMark make sure that data is available** **30 RESTORE 200+Skill*30 :REMark select required data line**
Description	A command which sets the data pointer to any selected **DATA** line. If none is specified, the first **DATA** line in the program is used
Format	**RESTORE** [<*line number*>]
Associated keywords	**DATA, READ**
Note	**RUN** does not perform an implicit **RESTORE**; thus it is possible to execute a program several times using different **DATA** lines

Type — Program control command

Purposes

1 To terminate a procedure and resume processing at the statement after the call to the procedure

2 To terminate a function and define the value of the expression which caused entry to the function

3 To terminate a subroutine and resume processing at the statement after the GOSUB call which caused entry to the subroutine

Examples

```
830 RETurn

540 RETurn (this_guess + last_guess) / 2
```

Description — A command causing termination of a procedure, function or subroutine. In the case of a function, the parameter defines the value of the function.

COMMENT

It is not compulsory to have a **RETURN** in a procedure. If processing reaches the **END DEF**ine of a procedure, the procedure returns automatically.

Format — **RETurn** [<*expression*>]

Associated keywords — **DEFine, END, FuNction, PROCedure**

Type	Program control command
Purpose	To allow a program to be resumed at the statement which was last executed, typically after an error condition
Example	**RETRY**
Description	A command which restarts a BASIC program, re-executing the last statement obeyed
Format	**RETRY**
Associated keywords	**CONTINUE**

Type — Mathematical function

Purpose — To generate a random number

Examples

```
30 chance = RND

PRINT RND(3 TO 27)
```

Description — If no parameters are supplied, the result is a floating point number in the exclusive range 0 to 1. If two parameters are supplied, the result is an integer in the range bounded by (and including) the parameters. If only one, the first is assumed to be zero. The second parameter (rounded if need be) must not be less than the first (rounded if need be)

Format — **RND** [([<numeric expression> **TO**]<*numeric expression*>)]

Associated keyword — **RANDOMISE**

Notes

1 If parameters are supplied, they must be in the range -32768 to +32767

2 Any parameter is rounded to an integer before being used and the range includes this integer

Type	BASIC command
Purpose	To commence execution of a stored program
Examples	**RUN** run from start **RUN 400** run from line 400
Description	A statement causing BASIC to execute the current program
Format	**RUN** [<*numeric expression*>]
Associated keywords	**LOAD, LRUN, MRUN, RESTORE**

Type	BASIC command
Purpose	To permit saving of programs
Examples	**SAVE prn** Print program on printer **SAVE mdv1_library TO 90** Save some lines of program
Description	A procedure which saves program listings to any device
Format	SAVE <*device specification*> [,<*line ranges*>]

Type — Memory access procedure

Purpose — To allow saving of areas of memory to a device

Example — **SBYTES MDV1_screen, 131072, 32768**

Description — A procedure allowing image dumps of areas of TONTO memory

Format — **SBYTES** *<file specification>*,*<memory address>*
,*<area length>*

Associated keywords — **LBYTES, PEEK, POKE, SEGMENT**

SDATE

Type — Clock procedure

Purpose — To allow the TONTO internal clock to be set to a specific time

Examples — **SDATE 1977,month,1,0,0,0**

120 SDATE 1984,6,14,10,28,0

Description — A procedure which updates the TONTO clock according to the value of its parameters

Format — **SDATE** *<year>, <month>, <day>, <hour>, <minute>, <second>*

Associated keywords — **ADATE, DATE, DATE$, DAY$**

Type	Store management function
Purpose	To allocate extra store to the BASIC user's program
Examples	150 LET Seg = SEGMENT(2)
Description	A function which returns the channel identifier of a free segment. The segment size is defined (in 512 byte memory blocks) by the supplied parameter
Format	**SEGMENT** (*<numeric expression>*)
Associated keywords	**PEEK, POKE, CALL, LBYTES, SBYTES**

Type BASIC command structure

Purpose To permit a multiway decision to be made according to the value of a variable

Example 230 SELect ON choice

Description A statement marking the start of a SELect...END SElect construct. SELect can be used in both long and short forms:

Long

Allows multiple actions to be selected depending on the value of a select variable. The select variable is the last item on the logical line. A series of BASIC statements follows, which is terminated by the next ON statement or by the END SELect statement, which allows a catch all which responds if no other select conditions are satisfied.

Example

```
 10 LET error_number = RND(1 TO 10)
 20 SELect ON error_number
 30  ON error_number = 1
 40   PRINT "Divide by zero"
 50   LET error_number = 0
 60  ON error_number = 2
 70   PRINT "File not found"
 80   LET error_number = 0
 90  ON error_number = 3 TO 5
100   PRINT "Microdrive file not found"
110   LET error_number = 0
120   ON error_number = REMAINDER
130   PRINT "Unknown error"
140   error_recovery
150 END SELect
```

Short

The short form of the **SELect** statement allows simple single line selections to be made. A sequence of BASIC statements follows on the same logical line as the **SELect** statement. If the condition defined in the **SELect** statement is satisfied, the sequence of BASIC statements is processed.

There is no END SELect statement.

Example

```
1     SELect ON test_data = 1 TO 10: PRINT "Answer within
      range"

2     SElect ON answer = 0.00001 TO 0.00005: PRINT "Accuracy
      OK"
```

COMMENT

The short form of the **SELect** statement allows ranges to be tested more easily than with an IF statement. Compare example 2 above with the corresponding IF statement.

Format — **SELect ON** *<numeric variable>*

Associated keywords — **END SELect, REMAINDER**

Note — The variable may not be a string variable or the value returned by a function

Type	Permanent store procedure
Purpose	To allow data to be stored and modified in the system's permanent store area
Examples	**SET_PSE 500, 'effort'** **340 SET_PSE last_used, DATE**
Description	A procedure which stores a string value of up to 255 bytes in the TONTO permanent store
Format	**SET_PSE** *<entry number>*,*<string value>*
Associated keywords	**DEL_PSE, PSE$**
Note	Do not set entries 0 to 255. These are reserved for system use

Type	Mathematical function
Purpose	To calculate the sine of an angle
Examples	**PRINT SIN (RAD(90))** **11 LET Erx = ABS(Angle - ATAN(SIN(Angle)/COS(Angle)))**
Description	A function returning the sine of its parameter which is in radian measure
Format	**SIN (**<*angle in radians*>**)**
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COS, COT, DEG, RAD, TAN**

Type	Mathematical function
Purpose	To calculate the square root of a non-negative number
Examples	**PRINT SQRT (3)** **LET C = SQRT (A*A + B*B)**
Description	A function returning the square root of its parameter
Format	**SQRT (**<*numeric expression*>**)** range > = 0

Type — BASIC operator

Purpose — Part of the **FOR...END FOR** loop construct. **STEP** indicates the control variable increment value

Example — **10 FOR I = 2 TO 100 STEP 2: PRINT I** Prints the even numbers from 2 to 100

Description — Part of the **FOR...END FOR** construct. **STEP** is optional

Format — **STEP** *<numeric expression>*

Associated keywords — **END FOR, EXIT, FOR**

Notes — This keyword never appears on its own

When **STEP** is omitted, an increment of 1 is used

Type — Program control command

Purpose — To terminate execution of a program and return BASIC to command interpreter mode

Example — **900 STOP**

Description — A procedure causing execution of a program to cease

Format — **STOP**

Associated keyword — **BYE**

Type	Mathematical function
Purpose	To calculate the tangent of an angle given in radians
Examples	100 **value = TAN (PI / 16)**
Description	A function returning the tangent of its parameter
Format	**TAN** (<*angle in radians*>)
Associated keywords	**ACOS, ACOT, ASIN, ATAN, COS, COT, DEG, RAD, SIN**

Type	BASIC command separator
Purpose	A keyword used within an **IF** construct to indicate the required course of action if the condition is **TRUE**
Example	**IF A>B THEN PRINT 'Bigger'**
Description	An optional part of the **IF...THEN...ELSE...END IF** construct
Format	**THEN** { <*statement*>}
Associated keywords	**IF, ELSE, END IF**
Note	This keyword never appears on its own

Type	BASIC separator
Purpose	To separate two numeric values in a range of numbers implying that all numbers in that range are to be considered
Examples	**FOR I = 1 TO 200 : PRINT I** **LET sub$ = whole$(4 TO 7)**
Description	A separator usable with numeric range expressions
Format	*<numeric value>* **TO** *<numeric value>*
Associated keywords	**FOR, SELect**

Type — Screen handling procedure

Purpose — To set or reset underlining of all subsequent characters output to the specified screen channel

Examples

```
UNDER 1                 Turn underlining on

100 OPEN #4,scr_480x20a0,0  :REMark title screen
105 CLS #4
110 CSIZE #4,3,1
115 PRINT #4,FILL$(" ",11);
120 UNDER #4,1
130 PRINT #4,"Generate Invoices"
```

Description — A procedure which changes the defined window attributes in respect of underlining

Format — **UNDER** [#*<channel number>*,]*<switch>*

where : *<switch>* = 0 underlining off
<switch> = 1 underlining on

VER$

Type — BASIC function

Purpose — To return a two character string indentifying the version of the interpreter

Examples

```
PRINT VER$

120 IF VER$ <> '02' THEN PRINT 'Wrong version'
```

Description — A function returning a fixed two character string as its result

Format — **VER$**

Type — Device control procedure

Purpose — To set a nominal page width of a device other than **CON** or **SCR**

Examples — **110 WIDTH #7,120 : PRINT a$! b$! c$**

Description — A statement controlling the overall output field width

Format — **WIDTH** [#<*channel number*>,]<*numeric expression*>

Notes — The current value of **WIDTH** has no effect other than in the treatment of ! in **PRINT** and **INPUT** statements. Following the separator !, the next item output will start on a new line if it would otherwise cause the current line to exceed the specified width for the current channel

Type	Screen handling procedure
Purpose	To permit the changing of position and size of a screen window
Examples	**WINDOW #2, 480, 80, 0, 120** Reduce the size of the listing window **10 WINDOW 60, 60, 0, 0**
Description	A procedure allowing the definition of a screen window. The coordinates are expressed in character multiples of pixel coordinates and are rounded to the nearest multiple of character size, which is 6 pixels wide and 10 pixels high. Thus coordinates of 22, 22 are rounded to 24, 20
Format	**WINDOW** [#<*channel number*>,]<*width*>,<*height*>,<*x*>, <*y*>
Associated keywords	**AT, CLS, CSIZE, INK, PAPER**

Type

BASIC operator

Purpose

To calculate the logical **exclusive OR** of two operands

Example

```
100 IF (bound_a < 0) XOR (bound_b < 0) THEN
110   PRINT "Range includs zero"
120 END IF
```

Description

The two operands are treated as truth values, zero being false, non-zero truth. The result is 0 (i.e. false) if both operands are false or both operands are true, 1 (i.e. true) if the two operands have different truth values. To form the bitwise XOR of two operands you should use the operator ^^

Format

<boolean expression> **XOR** *<boolean expression>*

Associated keywords

AND, NOT, OR

Appendix 1 TONTO BASIC Syntax

This appendix defines TONTO BASIC syntax using a form of the Backus-Naur Form (BNF).

Notes:

- ::= indicates that the expression on the left hand side of the ::= sign consists of zero or more items on the right hand side;

- | indicates that the items thus separated are alternatives; that is to say that one (and only one) of them must be present;

- { } denotes possible repetition of the enclosed symbols zero or more times; this means that

  A ::= B

  is only a simple form of

  A ::= <empty> | B | BB | BBB | .....

- the square brackets denote possible repetition of the enclosed symbols zero times or once only; this means that

  A ::= B [C] [D]

  is only a simpler form of

  A ::= B | BC | BD | BCD

- limits on values are specified in the form range x to y.

Note that the symbols used (::= | { } [and]) are meta-symbols belonging to BNF, and are not symbols of the BASIC language.

<angle in degrees> ::= *<numeric expression>*

<angle in radians> ::= *<numeric expression>*

<anything> ::= (No syntax; terminated by line-feed)

<area length> ::= *<integer expression>*

<arithmetic operator> ::= + | - | (| / | * | MOD | DIV | ^

<array> ::= *<variable>* | *<array element>*

<array element> ::= *<variable>* (*<expression>* { , *<expression>* })

<binary operator> ::= *<comparison operator>* | *<arithmetic operator>* | *<bitstring operator>* | *<boolean operator>* | *<string operator>*

<bitstring operator> ::= && | :: | ^^

<boolean expression> ::= *<integer expression>* range 0 to 1

<boolean operation> ::= OR | XOR | AND

<channel number> ::= *<integer expression>* range 0 to 15

<clock value> ::= *<integer expression>*

<colour code> ::= *<integer expression>* range 0 to 7

<column> ::= *<integer expression>*

<comparison operator> ::= = | == | < | > | >= | <= | <>

<condition> ::= *<boolean expression>*

<day> ::= *<integer value>* range 1 to 31

<device specification> ::= *<integer>* | *<string expression>*

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<dimension> ::= *<integer expression>*

<dimension number> ::= *<integer expression>*

<direct command> ::= *<BASIC command>*

<duration> ::= *<integer expression>* range 0 to 255

<enclosed string> ::= '{<any character except'>}'

<entry number> ::= *<integer expression>* range 0 to 65535

<expression> ::= *<term>* | (*<expression>*) | *<expression>* *<binary operator>* *<expression>*

<file specification> ::= *<device specification>*

<first line> ::= *<line number>*

<for range> ::= *<range>* [STEP *<numeric expression>*]

<frequency> ::= *<integer expression>* range 0 to 255

<function name> ::= *<variable>*

<function result> ::= *<function name>*[(*<expression>*{, *<expression>*})]

<height> ::= *<integer expression>* range 5 to 240

<height code> ::= *<integer expression>* range 0 to 1

<high bound> ::= *<numeric expression>*

<hour> ::= *<integer expression>* range 0 to 23

<identifier> ::= *<letter>*{*<letter>* | *<digit>* | *<underscore>*}

<increment> ::= *<integer expression>* range 1 to 32767

<integer expression> ::= *<expression>* range -32768 to 32768 unless explicitly stated otherwise

<integer variable> ::= *<integer>*%

<IO list> ::= [*<expression>*{*<print separator>* *<expression>*}]

<last line> ::= *<line number>*

<letter> ::= A|B|C|D|E .....|Z|a|b|c|d|e|.....|z

```
<line number>            ::= <expression> range 1 to 32767

<line number list>       ::= <line number>{,<line number>}

<line range>             ::= [<line number>]
                             [TO <line number>]
                             [<line number>TO]

<line ranges>            ::= <line range>{ ,<line range>}

<loop identifier>        ::= <identifier>

<loop variable>          ::= <identifier>

<low bound>              ::= <numeric expression>

<memory address>         ::= <expression>|<segment identifier>,
                             <offset>

<minute>                 ::= <integer expression> range 0 to 59

<month>                  ::= <integer expression> range 1 to 12

<new first line>         ::= <line number>

<numeric expression>     ::= <expression>

<numeric variable>       ::= <variable>

<offset>                 ::= <expression> range 0 to segment_
                             size -1

<one or two character    ::= <string expression>
string>

<TONTO program name>     ::= <identifier>

<parameter>              ::= <identifier>

<print separator>        ::= !|,|;|\

<procedure name>         ::= <identifier>

<quoted string>          ::= "{<any character except">}"

<range>                  ::= <low bound  [TO <high bound>]

<register value>         ::= <integer expression>

<repeat count>           ::= <integer expression>
```

<*required values*> ::= <*for range*>{,<*for range*>}

<*row*> ::= <*integer expression*>

<*search key*> ::= <*integer expression*> range 0 to 3

<*seconds*> ::= <*integer expression*> range 0 to 59

<*section*> ::= <*integer expression*>

<*segment identifier*> ::= <*numeric expression*>

<*select list*> ::= <*low bound*> [TO <*high bound*>] |
<*low bound*> TO | TO <*high bound*>

<*slice*> ::= <*variable*> (<*select list*>)

<*statement*> ::= <*BASIC statement*>

<*string*> ::= <*enclosed string*> <*quoted string*>

<*string expression*> ::= <*expression*>

<*string operator*> ::= &|INSTR

<*string value*> ::= <*string expression*>

<*string variable*> ::= <*identifier*>

<*switch*> ::= <*integer expression*> range 0 to 1

<*term*> ::= <*value*>|<*unary operaytor*><*value*>

<*unary operator*> ::= +|-|~~|NOT

<*underscore*> ::= _

<*unsigned constant*> ::= <*unsigned integer*>|<*unsigned real*>|
<*string constant*>

<*unsigned decimal*> ::= <*unsigned integer*>|<*unsigned
integer*>.[<*unsigned integer*>]|
.<*unsigned integer*>

<*unsigned integer*> ::= <*digit*>{<*digit*>}

<*unsigned real*> ::= <*unsigned decimal*>|<*unsigned
decimal*>[E +|-] <*unsigned
integer*>

<*untyped variable*> ::= <*identifier*>

<value>	::=	*<unsigned constant>* | *<variable>* | *<array element>* | *<slice>* | *<function result>*
<variable>	::=	*<untyped variable>* | *<integer variable>* | *<string variable>*
<wait period>	::=	*<integer expression>*
<width>	::=	*<integer expression>* range 3 to 480
<width code>	::=	*<integer expression>* range 0 to 3
<x>	::=	*<expression>* range 0 to 474
<y>	::=	*<expression>* range 0 to 230
<year>	::=	*<integer expression>* range 1970 to 2069

Appendix 2 TONTO BASIC reserved words

Reserved words are words which have a particular meaning for TONTO BASIC. Use of them in the wrong context (for example, as identifiers) is likely to cause errors or unpredictable results.

The full list of reserved words is given below; it includes all current keywords, all keywords reserved for future use and a few other words which are meaningful to BASIC.

Where a reserved word is printed in partly upper and partly lower case, it means that the upper case part is itself a reserved word, as are all words starting with the upper case part followed by consecutive letters from the lower case part, up to the whole word. Thus, in the case of ERRor,

```
ERR
ERRo
ERRor
```

are all reserved words. The actual case used at the TONTO keyboard is of course irrelevant, except that case in filenames is meaningful.

ABS	CODE	ELSE	INT
ACOS	CONTINUE	END	KEYROW
ACOT	COPY	EOF	LBYTES
ADATE	COPY_N	ERLin	LEN
ALL	COS	ERNum	LET
AND	COT	ERRor	LINE
ARC	CSIZE	EXEC	LINE_R
ARC_R	CURSOR	EXEC_W	LIST
ASIN	DATA	EXIT	LN
AT	DATE	EXP	LOAD
ATAN	DATES	FILL	LOCAL
AUTO	DAYS	FILLS	LOG10
BAUD	DEFine	FLASH	LRUN
BEEP	DEG	FOR	MERGE
BEEPING	DELETE	FORMAT	MISTake
BLOCK	DEL_PSE	FUNction	MOD
BORDER	DIM	GO	MODE
BYE	DIMN	GOSUB	MOVE
CALL	DIR	GOTO	MRUN
CHR$	DIV	IF	NET
CIRCLE	DLINE	INCLUDE	NEW
CIRCLE_R	DRAW	INK	NEXT
CLEAR	EDIT	INKEY$	NOT
CLOSE	ELLIPSE	INPUT	ON
CLS	ELLIPSE_R	INSTR	OPEN
OPEN_IN	POKE_L	RETRY	STOP

OPEN_NEW	POKE_W	RETurn	STRIP
OR	PRINT	RND	SUB
OVER	PROCedure	RUN	TAN
PAN	PUBLISH	SAVE	THEN
PAPER	PSE$	SBYTES	TO
PAUSE	RAD	SCALE	TRACE
PEEK	RANDOMISE	SCROLL	TURN
PEEK_L	READ	SDATE	TURNTO
PEEK_W	RECOL	SEGMENT	UNDER
PENDOWN	REMAINDER	SELECT	VERS
PENUP	REMark	SET_PSE	WHEN
PI	RENUM	SEXEC	WIDTH
POINT	REPeat	SIN	WINDOW
POINT_R	REPORT	SQRT	XOR
POKE	RESTORE	STEP	

Appendix 3 Transferring data between applications

BASIC provides the ability to create, read and write files on microdrive cartridges. Other applications, notably those in XChange, also create and process such files, and it is therefore possible to exchange data between these applications via microdrive files, provided both partners in the exchange understand the layout and content of the data.

BASIC can be used to create two kinds of file. A file written using SAVE or PRINT statements consists of lines. Each line is a string of characters terminated by the line-feed character. The last line may be followed by a single character with the code 26 (CTRL/Z). A file written using SBYTES is a binary copy of the specified area.

BASIC normally deals with file names that have the same form as a BASIC identifier. For example

```
SAVE mdv1_Program_1
```

A filename may also be specified as a string expression, thus

```
SAVE "mdv1_Program_1"
f$ = "mdv1_Program_1" : SAVE f$
p$ = "Program_"       : SAVE "mdv1_" & p$ & 1
```

This method of naming files may be necessary when dealing with files created by, or to be read by, non-BASIC applications. In particular, XChange filenames have the form

```
<identifier>.<extension>
```

and must always be identified to BASIC in the string form, for example,

```
OPEN_NEW #6,"mdv1_XCfile.exp"    : REM create a file for
                                   input to XChange
COPY "mdv2_datafile.exp" TO scr : Copy an XChange file to
                                  the screen
```

Index

Item code: 983081
Issue: 1(9/84)
Publication number TPU 12C