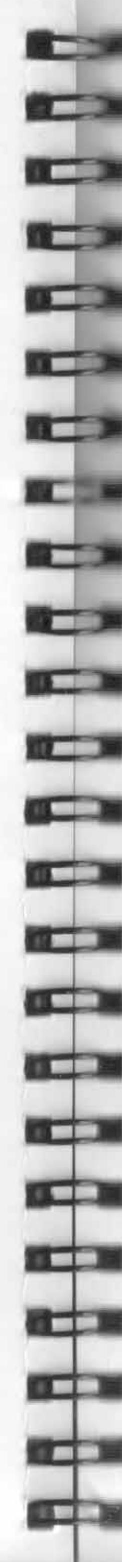
CAMPUTERS

LYNX

USER MANUAL



CAMPUTERS
LYNX
USER MANUAL

Written and illustrated by Sue Jansons
Cover design by Michael Peters & Partners
Typeset in 9pt Rockwell Light and 7pt OCR-B by Wordsmiths
Printed by Libra Press Ltd
Produced by Lang Communications Ltd
Published by Camputers Ltd, 33A Bridge Street, Cambridge CB2 1UW

CONTENTS

1 SETTING UP THE COMPUTER
Cassette settings 1
Before you begin 2
How to use the manual 3
Colours 4

2 THE KEYBOARD
The prompt, the cursor, the keys 5
Memory, MEM 6

3 THE COMPUTER AS A CALCULATOR
Calculator mode 7
CLS 7
Addition, subtraction, etc. 7
RAD, DEG, sines, cosines, tangents 8
Logarithms, LOG, ANTILOG 8
Random numbers, RAND, RND, RANDOM 8
The algebraic hierarchy 9

4 STARTING TO PROGRAM
Line numbers 11
AUTO 11
Maximum line length 12
RUN 12
PRINT 13
PRINT TAB 14
Variables 15
LET 15
SWAP 15
String variables 16
INPUT 17

5 LOOPING
GOTO 19
PAUSE 20
FOR...NEXT 20
END 23

6 MAKING DECISIONS
IF...THEN 24
Relational operators 25
Logical operators 25
IF...THEN with strings 26
IF...THEN, ELSE 27
IF...THEN with GOTO 28

7 MORE ABOUT STRINGS
DIM 29

CHR$ 29
KEYN and KEY$, GETN and GET$ 30
LEFT$, RIGHT$, MID$ 30-1
VAL 31
ASC 31
UPC$ 31
LEN 31
String expressions 32

8 EDITING
REM 33
LIST 34
DEL 34
ESCAPE and CONT 34
STOP 34
TRACE 35
SPEED 35
RENUM 35
NEW 35
Editing 36

9 STORING and LOADING PROGRAMS
SAVE 37
VERIFY 37
LOAD 38
APPEND 38
MLOAD 39
TAPE 39

10 MORE VARIABLES
Arrays, DIM 40
READ, DATA, RESTORE 41

11 STRUCTURING COMPLEX PROGRAMS
Subroutines: GOSUB, RETURN 44
PROCEDURES 46
REPEAT...UNTIL 47
WHILE...WEND 48
TRUE and FALSE 49
ERROR 49

12 FURTHER MATHS
Scientific notation 50
ROUND and TRAIL 50
INT, FRAC, ABS, SGN 51
ARCSIN, ARCOS, ARCTAN 51
INF 51
DIV 51
MOD 52
Factorials 52
Exponentials and natural logs 52

13 THE PRINTER
LLIST 53
LPRINT 53
LINK 53

14 GRAPHICS AND SOUND
The colours, INK and PAPER 54
The graphics characters and codes 55
PROTECT 55
High resolution graphics 57
The screen 57
The graphics cursor 57
MOVE 57
DRAW 58
DOT 58
PLOT 58
WINDOW and PRINT@ 59
POS and VPOS 60
BEEP 60
Changing the cursor, CCHAR and CFR 61
Control codes, PRINT CHR$, VDU 62
User defined graphics 62
ALPHA, GRAPHIC, LETTER, BIN 63

15 WHAT IS MACHINE CODE? 66

16 MACHINE CODE
and & 69
PEEK and DPEEK 69
POKE and DPOKE 70
CODE, LCTN, CALL and HL 70
HIMEM and RESERVE 71
Binary operators 71
INPT and OUT 71
SOUND 72
The monitor 72

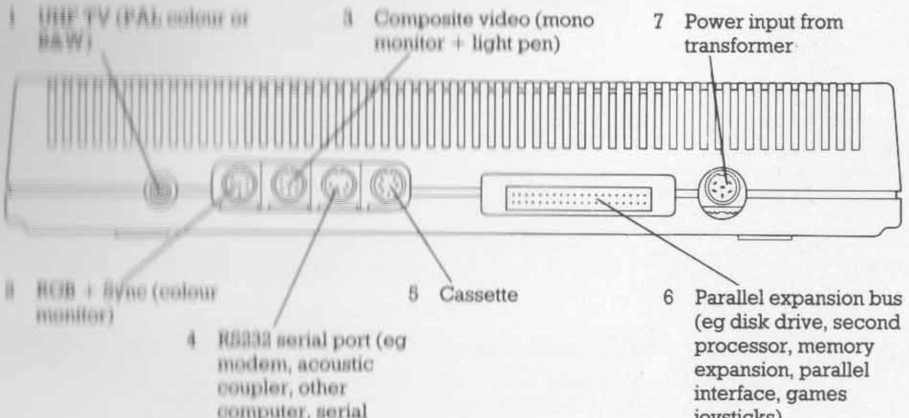
APPENDIX 1: ERROR MESSAGES 78
APPENDIX 2: SHORTHAND 80
APPENDIX 3: ASCII CODES 81
APPENDIX 4: EXTERNAL CONNECTIONS TO THE LYNX 83
APPENDIX 5: SUMMARY OF LYNX BASIC 84

Chapter 1: SETTING UP THE COMPUTER

First examine

1. The Lynx computer/keyboard, particularly the sockets at the back.
2. The video lead (which has an aerial plug at one end). This is used to connect the Lynx to a television set.
3. The cassette lead (which has a DIN plug on one end and three jack plugs on the other end).
4. The power supply.

Now examine the following diagram.



It is important to connect the computer up in the right order, so read the following instructions right through before you start.

First, take the video lead. Plug the aerial plug into the aerial socket of the television set, and the other end into the socket (marked 1 on the diagram above) on the computer.

Next, if you're planning to load pre-recorded programs into the computer or to record your own programs, you'll need to connect a cassette recorder with the cassette lead. At the cassette recorder end it has three jack plugs: a grey one, a thick black one and a thin black one. Plug the grey jack into the socket marked EAR, the thick black jack into the socket marked MIC, and the thin black jack into the REMOTE socket. (If you have no remote socket, it is alright to leave it hanging free). Then plug the DIN end into the socket marked 5 on the diagram above.

Then plug the DIN plug on the power supply into the socket marked 7 on the diagram. Do not force it. *BE VERY CAREFUL NOT TO PLUG IT IN UPSIDE DOWN.*

Turn on the television and the cassette recorder.

Now plug the power supply into the mains.

The computer will make a (pleasant) beeping noise. Tune the television to channel 36. A Lynx logo will appear in the top left-hand corner.

You are now ready to go!

It is wise to connect to the mains last, and disconnect from the mains first, so to disconnect the computer follow the above in reverse order.

CASSETTE PLAYER SETTINGS

(Come back to this later!)

To record programs successfully you will need to have your cassette player on the correct settings.

If there is a tone control on your player turn it to HIGH.

You will have to experiment to find the best volume setting, because it will depend on your particular machine. When you have played with the Lynx a little, but *before you have a program you want to record*, try typing something like this into the computer:

```
100 REM THIS IS A TEST [RETURN]
110 REM [RETURN]
120 REM [RETURN]
130 REM [RETURN]
```

Set the cassette player to a volume near the middle of its range, then – following the instructions in Chapter 9 – try saving, verifying and loading your dummy program. If the program is corrupted by the process, try it with a slightly higher volume, then a slightly lower one, until you are successful. If the program is not saved at all, again try with different volumes.

Once you can save and load, you can find the most reliable setting by finding the highest successful volume, and the lowest, and setting it in the middle.

BEFORE YOU BEGIN

The 'computer' part of a computer system differs slightly from design to design, but always consists of a microprocessor and two different types of memory: ROM (read-only memory) and RAM (random access memory).

The microprocessor is the 'brain' of the machine, but it cannot process material without being first told how to do it. Its instructions are contained in the ROM (which is a permanent storage area: programs stored there can only be read, not altered or erased) and vary from computer to computer. But the end result is similar in all microcomputers, and forms a computer **language**.

The RAM is the computer's working memory. It is here that programs, such as word processors, Space Invaders, and any programs you write yourself, are stored. This memory is erasable: its contents are destroyed when the computer is switched off; and during use, it can be cleared using special commands.

The physical parts of the computer, the chips, the resistors and capacitors, the case, and so on, are called the **hardware**; programs are called **software**. So the working computer consists both of hardware and some permanent software.

The computer is linked to the outside world by various devices, which collectively are called **peripherals**. These can include a keyboard; a screen display of some kind, a monitor or television set; a cassette player; a disk drive; and a printer.

You need to be able to enter information into the computer's memory, and there are several peripherals which enable this. The most immediate is the keyboard, which allows you to type material directly into the computer.

It is also possible to load stored information into memory, either from cassette or floppy disk.

The computer needs to be able to communicate its results back to you: these can be shown immediately on the screen display; or can be made into a permanent copy (**hard copy**) on a printer; or can be stored for future re-use on cassette or disk.

To see why a computer is so versatile, we need to consider the nature of a program. A program is a series of instructions arranged in a fixed order. The computer can perform a limited number of simple operations; it is the programmer's task to break down complex problems into a series of simple tasks ready for the computer to process. The computer can perform these tasks at great speed and with great accuracy – and unlike humans it never tires of repeating an operation, and it never loses concentration. But the computer's versatility is really the result of the programmer's skill and ingenuity.

The programmer needs to be able to do two things: first to analyse whatever problem he has set himself – that is, work out the series of simple steps which bring him his answer; then **encode** this solution into the the language used by the computer. In practice, the two stages are closely linked, because the way the programmer solves his problem will be influenced by the characteristics and capabilities of the particular language involved.

Most microcomputers use a language called Basic, which is easy to learn and to use because its command words and its structure are similar to ordinary English. (In fact the letters BASIC stand for Beginners' All-purpose Symbolic Instruction Code). As you become more and more familiar with it you will find yourself able to solve more and more complex problems.

And you will probably find that, quite apart from allowing you to use the computer, learning to program will stimulate you to see things in new ways, and to develop new ideas.

HOW TO USE THE MANUAL

IF YOU ALREADY KNOW BASIC:

your best strategy will probably be to consult the Summary at the end of the manual, and chapters 11 (Structuring complex programs), 14 (Graphics) and 16 (Machine code).

IF YOU DON'T KNOW BASIC:

this manual was written for you! Its aim is to take you carefully through the features of Lynx Basic.

It begins slowly and simply, but assumes that as you learn more about Basic, you will want to work a little faster.

Remember that programming is about solving problems. So approach the manual bearing in mind the sorts of things you want to be able to do, and look at the things you learn in that light. With this approach in mind, most chapters include a section of Examples and Ideas at the end.

Before you begin, you might like to experiment with colour, which is, after all, one of the Lynx's most exciting features.

The Lynx has eight colours

```
0-BLACK
1-BLUE
2-RED
3-MAGENTA
4-GREEN
5-CYAN
6-YELLOW
7-WHITE
```

When you switch it on, the Lynx is programmed to write in white on a black background, but you can work in any colours you like.

You can change the background colour of the screen – to red, for example – by typing either

```
PAPER RED
```

and pressing the key marked *[RETURN]*, or

```
PAPER 2
```

and pressing *[RETURN]*.

And you can change the 'ink' colour – to black, say – by typing

```
INK BLACK [RETURN]
```

or

```
INK 0 [RETURN]
```

Whatever you type now will appear as black print on a red screen – try it. If you want to clear the screen and start again, type:

```
CLS [RETURN]
```

Remember that if you make ink and paper the same colour, you will not be able to see the writing on the screen (in fact, you may think that the computer has broken down!), but the computer will still recognise and interpret anything you type in.

Chapter 2: THE KEYBOARD

The computer has a keyboard very similar to that of an ordinary typewriter, and the letters and numbers are arranged in the same order as on a standard keyboard, with two shift keys and a shift lock. But there are some important additional keys, and some of the familiar keys have new functions.

When the computer is 'ready' for operation, it displays a prompt symbol > and marks your position on the screen with a flashing block, which is called the **cursor**. This shows you where the character you are typing will appear on the screen. As you type text into the computer it is printed on the screen, and the cursor moves accordingly. When the screen is full, the computer starts writing from the top again. If the text is part of a program it is stored in the computer's memory.

If you hold a key pressed down, it will repeat automatically.

SPECIAL KEYS

The most important of the additional keys is *[RETURN]*. By pressing the *[RETURN]* key you tell the computer to process the text you have typed in. The computer will know whether the text makes sense or not: if it doesn't, the appropriate error message will be displayed on the screen. (For a complete list of possible error messages, see Appendix 1). If you have typed in a **program line**, the computer will store it until it is told to run the program. If it has been given an instruction which is not part of a program – a calculation, for example, it will execute it immediately. The important thing to remember is that the computer will not process anything until you tell it to by pressing *[RETURN]*.

The *[DELETE]* key allows you to delete characters by backspacing over them.

The left and right arrow keys move the cursor in the appropriate directions; this will not alter the text.

The *[CONTROL]* key is similar to a shift key: for its uses, see Chapter 8 on Editing, and Chapter 14 on Graphics.

The *[ESC]* key (short for ESCAPE) allows you to stop a program whilst it is running without damaging the program. This is explained more fully in Chapter 8.

The *[BREAK]* key is used by certain specialised programs, but is ignored when the Lynx is running normally.

The * has a special meaning on a computer: it is used instead of x to mean multiply, to prevent confusion. Two **s means 'raised to the power of', so 4 squared is written 4**2, 6 cubed is 6**3.

The / is used to mean divide.

As well as meaning subtract, the - can be used to negate numbers or variables.

The . (full stop) can be used as a decimal point.

The < and > symbols stand for 'less than' and 'greater than' respectively. They are used in situations where you want the computer to make decisions. See Chapter 6.

There is no π symbol on the keyboard: you enter π by typing PI.

The Lynx has a **shorthand** facility to help you type in programs – see Appendix 2.

Commands can be typed in either upper or lower case or a combination of both: the computer will convert them all to upper case when it lists the program.

MEMORY

You may need to know something about the terms used to describe different quantities of memory.

A **bit** is the smallest unit of memory: it stores one single digit which, because of the way a computer works, can either be a 0 or a 1. (Hence its name, which is short for **bi**nary digi**t**).

Most microprocessors work in units of eight bits; eight bits are called a **byte**.

1024 bytes are called a **kilobyte**, which is abbreviated to **K** (1024 is a 'binary thousand', 2 raised to the power of 10).

The Lynx has 48K of memory, which is 49152 bytes.

MEM

If at any time you want to know how much memory you have left, you can type `MEM` (and press `[RETURN]`), and the computer will display (roughly) the number of bytes left.

Chapter 3: THE COMPUTER AS A CALCULATOR

You can use the computer for calculations both inside and outside programs, and the Lynx uses a format similar to that used by a pocket calculator: it has a special 'calculator mode'.

The computer also has certain **functions** stored in its memory. A function is a ready-made set of instructions for carrying out a calculation: like finding the logarithm of a number. Functions have this format:

```
function name(X)
```

X represents the thing you want to process, which must be placed in brackets. It can be a number, a **variable** (see Chapter 4), or an **expression** (an expression is any sequence of numbers and symbols which is intended to calculate a value, like 3+4). The thing the function is asked to process is called its **argument**. In this chapter we will look at some of the more commonly used functions.

Before you try them out, here's an instruction you may find useful:

CLS

If you want to clear the screen, type `CLS [RETURN]`. This also 'homes' the cursor – that is, moves it to the top left-hand corner of the screen.

Now you know how to unclutter the screen, you can try some calculations. You can add numbers together by typing them in like this:

```
4+2
```

then pressing `[RETURN]`. The computer will immediately print the answer. Similarly, to subtract, type:

```
4-2 [RETURN]
```

To multiply, type:

```
4*2 [RETURN]
```

And to divide, type:

```
4/2 [RETURN]
```

You can square a number (multiply it by itself) like this:

```
4**2 [RETURN]
```

The `**` stands for 'raised to the power of': the four is the number to be processed, the

two shows that it is to be multiplied together twice. You can cube a number (eg 4*4*4) by 'raising to the power' of three:

```
4**3 [RETURN]
```

You can raise any **positive** number to the power of any number.

For calculating a square root of a number (the number which, when multiplied by itself gives the first number) there is a special function, `SQR`
It is used like this:

```
SQR(16) [RETURN]
```

For calculating the cube root (which when multiplied together three times gives the number) or any higher root, use the this formula:

```
X**(1/n)
```

`X` represents the number you want to calculate the root of, and `n` represents the type of root you want. Again, note the use of brackets. So, if you wanted the fifth root of 100 (the number which when multiplied by itself five times gives 100), you would type:

```
100**(1/5) [RETURN]
```

You can only find the roots of **positive** numbers.

The computer has π stored in memory with the value of 3.1415927. There is no π symbol on the keyboard; to use π you must type `PI`.

ANGLES, SINES, COSINES and TANGENTS

The computer can calculate the **sine**, **cosine** and the **tangent** of an angle.

The angle must be given in **radians**. The Lynx can convert an angle in degrees to one in radians, or an angle in radians to one in degrees (there are 2*(PI) radians – and 360 degrees – in a full turn), like this:

```
DEG (angle in radians)
RAD (angle in degrees)
```

The sine, cosine and tangent functions can be used like this:

```
SIN(X)
COS(X)
TAN(X)
```

where `X` represents the angle in radians.

LOGARITHMS

The computer also calculates logarithms and antilogarithms. To multiply numbers together using logs, find the log of each number, add them together, then find the antilog of the answer. To divide using logs, find the log of each number, subtract them, then find the antilog of the answer.

```
LOG(X)
```

gives the the log of a number,

```
ANTILOG(X)
```

gives the antilog.

RANDOM NUMBERS

The Lynx has two functions which generate (pseudo) random numbers: `RAND` and `RND`.

```
RAND (X)
```

will give you a random number between 0 and X-1 (it will give you one of X possible numbers). If you want a number between 1 and X, just add 1, like this:

```
RAND (X)+1
```


So to imitate a die, you would use:

```
RAND (6)+1
```

`RND` gives you a random number between 0 and 1, including 0, but not 1, and you will probably have to process it to bring it into the range you want. `RND` is most useful in statistical operations.

The numbers these two functions generate are not truly random, because they are created by the same formulae, using the same initial values, every time the computer is used. But for most purposes their randomness is adequate.

If you want to use random numbers in a program, and to ensure that they will be different every time the program is run, you can insert the command `RANDOM` at the beginning of the program. This will reset the initial value of the random number generator each time.

CALCULATING DURING A PROGRAM

If you have a program running on the computer, you can halt it for a while using `[ESC]` (see Chapter 6) and carry out calculations in calculator mode. The program remains intact, stored in memory. You can restart the program again, using `CONT [RETURN]`.

When you use them inside a program, calculations have the same format as they have outside.

THE ORDER OF CALCULATIONS

When making calculations, the computer observes an **algebraic hierarchy**: it has been programmed to perform calculations in strict order. This is achieved by allocating a priority – a number between 0 and 22 – to each operation. The computer executes them in numerical order, highest first. Given a calculation including several different operations, it will

– first evaluate **functions**;

– then it will calculate **powers** (like squares or cubes);

– next, it will give a **negative** value to any numbers you have marked with a minus sign (-);

– then it will carry out **multiplication** and **division**: they have the same priority and so are executed in order from left to right;

– finally, it will carry out **addition** and **subtraction**, which also have the same priority and are also executed from left to right.

It is important for you to be familiar with this order of priorities because you must construct calculations accordingly, otherwise the computer may not make the calculation you want, but a completely different one. For example,

```
5+6*10
```

will be calculated like this:

6*10=60
5+60=65

when you might have expected the answer to be 110,or

5+6=11
11*10=110

You can alter the priority of an operation by placing it in brackets: this will give it priority over any other operation. So,

```
(5+6)*10
```

would be calculated

5+6=11
11*10=110

Being able to change priorities in this way can be very useful. Otherwise calculations like

```
(a+b)*(c+d+e)
```

would involve complex arrangements of operations:

a*c+a*d+a*e+b*c+b*d+b*e

The computer has several other important functions which are more specialised than those discussed in this section, including **natural logs**, and **factorials**. These will be discussed in Chapter 12.

Chapter 4: STARTING TO PROGRAM

LINE NUMBERS

A program is a sequence of instructions. In Basic, this sequence is determined by **line numbers**. Each program line begins with a number, and is entered into the computer when you press *[RETURN]*. The computer **arranges** and **executes** the lines in numerical order.

It is usual to start numbering with a fairly high number, say 100, and to increase each subsequent number by 10. You may need to add more lines later, and this should ensure that you have enough space. Even experienced programmers do this!.

The line numbers do not have to have regular increases between them, and the lines do not have to be typed into the computer in numerical order.

The computer has two modes: immediate mode and program mode. A line number tells the computer that the instruction following it is part of a program. Try typing this in:

```
PRINT "LYNX "
```

Press *[RETURN]*. The computer obeys the command immediately. Now try this:

```
10 PRINT "LYNX "
```

Again press *[RETURN]*. This time the computer does not obey immediately: instead it stores the line as part of a program.

You may like to add another line to your program:

```
20 GOTO 10    [RETURN]
```

The computer must be told to execute the program. Type:

```
RUN [RETURN]
```

The computer will obey immediately, and fill the screen full of 'LYNX's. To stop the program running, press the *[ESC]* key.

If you wanted to start it running again, you could type either

```
CONT [RETURN]
```

which would start it up from the point it stopped at, or

```
RUN [RETURN]
```

which would restart it from the beginning.

If you want to erase the program from the computer's memory, type `NEW` *[RETURN]*.

We will explore `PRINT`, `GOTO`, `RUN`, *[ESC]*, `CONT`, and `NEW` more fully later.

AUTO

You can ask the computer to put the line numbers in for you, by typing `AUTO` *[RETURN]*. You can use `AUTO` either before you begin typing in a program or during typing; then, when you press *[RETURN]*, the computer will write in the number of the following line for you.

You can tell the computer which number to begin from, and the rate of increase you want, like this:

```
AUTO 1000,100 [RETURN]
```

In this case numbering would begin with line 1000 and each subsequent number would increase by 100. Otherwise, the computer automatically begins numbering at

100 and increases by 10. You can just specify the starting number,

```
AUTO 1000 [RETURN]
```

and the computer will increase line numbers by 10.

When you want to stop `AUTO`, press *[RETURN]*.

Certain mistakes – mis-spelling a command, for example – will make a line unintelligible to the computer, and it will respond by displaying an error message. If you make an error whilst using `AUTO`, the computer will give you the error message, then print the same line number again, allowing you to retype the line.

When you overtype a line – that is, type in a line using a line number which already exists, perhaps as the remains of an old program – *the new line replaces the old*. If you are using `AUTO` and have asked for numbers which already exist, the computer will warn you by printing a ! after the number as it prints it up. If you do not want to overtype the line, you can keep the earlier version by pressing *[RETURN]*. You can then reset AUTO, specifying different numbers.

You can use the computer in calculator mode whilst in `AUTO`. For example, if the computer gives you

```
1000
```

and you type 4+4 *[RETURN]*

the computer will display

```
8
1000
```

and allow you to type in line 1000.

Certain Basic commands (like the `GOTO` in our Lynx program) use line numbers to redirect the flow of the program *against* numerical order. This is another important function of line numbers.

A special feature of Lynx Basic is that it allows you to have line numbers which are not whole numbers, like

```
10.5
```

or even

```
10.23656
```

which means that, should you run out of space between whole numbers but still need to insert more lines, you can do so without having to attempt to renumber your program. It also means that you will not have to resort to bad programming practices like using `GOTO` to direct the computer to bits of program you have had to write elsewhere.

MAXIMUM LINE LENGTH

On this computer there is a *maximum of 240 characters* allowed on any program line.

RUN

As we saw earlier, when you have typed in your program, it is stored until you tell the computer to execute it. You do this by typing `RUN` *[RETURN]*.

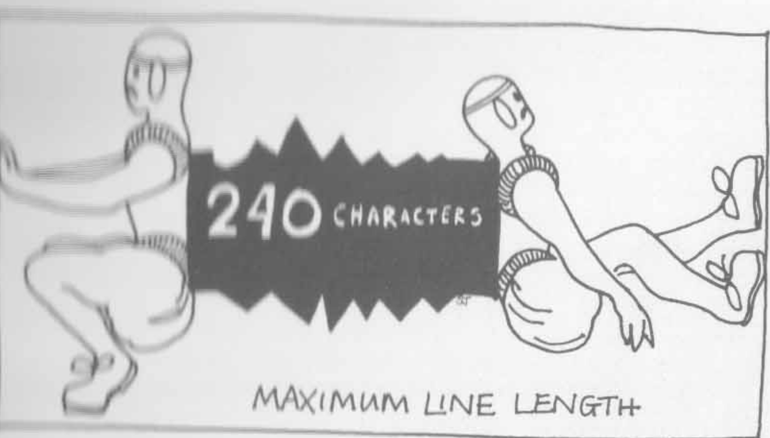
You can tell the computer to begin running from a specific point in the program by adding a line number to the command. For example:

```
RUN 100 [RETURN]
```


In this case the computer will ignore any line with a number lower than 100, and start executing from line 100. If any later part of the program sends it back to the earlier lines, however, it will execute them, and they will not have been impaired by the `RUN 100` command.

A program remains stored in the computer's memory until you erase it by using `NEW` (see Chapter 8), or by switching the computer off. Until then, it can run again and again.

When it runs a program, the computer has to keep track of where it is, it has to store the values of variables, and so on. This information is all left behind in the computer's memory when the program has finished running. As well as telling the computer to execute the program, `RUN` cleans all this debris away.



More about PRINT

As we have seen, the print command tells the computer to display information on the screen, and can be used inside or outside a program.

This information can be either a **string**, or a number or a **variable**. Let's look at strings first.

'String' is just the name for any collection of characters. It could be a name, or a unit, like 'inches', or an instruction like:

```
Would you like to play the game again?
```

On the Lynx strings are normally limited to 16 characters, including any spaces. But you can use strings up to 127 characters long by **dimensioning** them – which is explained in Chapter 7.

If you want to print a string, the material must be placed in inverted commas:

```
10 PRINT"I am a LYNX." [RETURN]
```

If you want to run this, type `RUN` *[RETURN]*.

The computer does not read anything contained in the inverted commas. It can be in English, in German, or gibberish: the computer will simply obey its instructions and print it on the screen.

A **variable** is a symbol which represents a numerical value. When you ask the computer to print a variable it gives you its present value (for a full explanation of variables see later in this chapter).

If you want to print a number or a variable, you do not use inverted commas:

```
10 LET A=0 [RETURN]
20 PRINT A [RETURN]
RUN [RETURN].
```

If you had put inverted commas around the A by mistake, the computer would have printed a letter A rather than the **value** of **variable** A.

You can combine the two types of print statement on one line. Retype line 20 as:

```
20 PRINT "The answer to the problem is ";A;
" inches."
```

and RUN [RETURN] the program again.

You may have noticed the semi-colons which separate the two types of print statement in the line above. These tell the computer how much space to leave before printing the material that follows.

A semi-colon tells it to leave no space, so be careful to insert any spaces you may need into the string in the inverted commas, like the space after is.

The computer's screen is divided (invisibly to the naked eye!) into 40 columns – you can print 40 characters across the screen. These columns are divided into 5 larger columns, each 8 characters wide. A comma tells the computer to move (**tab**) across to the beginning of the next column before printing.

If you combine the two types of print statement you need to place one or other of these symbols, called **delimiters**, between them. If you forget them, the computer will give you an error message.

If there is no delimiter at the end of a print statement, then any later printing will start at the beginning of the next line; if you add a delimiter, later printing will start either immediately after the previous material, or tabbed across from it.

This short program should show you the difference.

```
10 PRINT "This","is","an","example","of","commas" [RETURN]
20 PRINT "This ";"is ";"an ";"example ";"of ";
"semi-colons" [RETURN]
30 PRINT "This should join on to"; [RETURN]
40 PRINT " this." [RETURN]
RUN [RETURN]
```

When you want to erase the program, remember NEW [RETURN].

PRINT TAB

PRINT TAB allows you to select any one of the 40 columns and tell the computer to begin printing from there, like this:

```
PRINT TAB column number; material to be printed
```

or

```
PRINT material; TAB 20; material
```

Note that you must insert a semi-colon between the TAB command and the material to be printed.

Try this:

```
10 PRINT TAB 15;"I am a LYNX" [RETURN]
20 GOTO 10 [RETURN]
RUN [RETURN]
```

Remember that although the program forms a continuous loop you can stop it at any time using [ESC].

TAB is useful whenever you want to position text accurately on the screen.

VARIABLES

Type NEW [RETURN] to clear the computer's memory, then try typing in the following program:

```
10 LET a=0 [RETURN]
20 LET a=a+1 [RETURN]
30 PRINT a;" "; [RETURN]
40 PAUSE 5000 [RETURN]
50 GOTO 20 [RETURN]
RUN [RETURN]
```

The a in the program above is a **variable**, a label referring to a particular location in memory. The information stored in this location varies as the program runs: the first line of the program tells the computer to store a value of 0 in it. In the next line it is told to process the value of **a** by adding 1 to it: so, the the value of **a** becomes 1, then 2, and so on. Line 30 tells the computer to display the present value of **a**, followed by a space, on the screen. The last line tells it to repeat the whole process again, and it will do so again and again until stopped by [ESC].

The important thing to notice is that although the numerical value of **a** changes as the program runs, the number it represents always plays the same part in the program; it may be easiest to think of a variable as a symbol representing a number which changes in value because it is processed as the program runs.

A variable must be represented by a single character, and your choice of characters is restricted to the letters of the alphabet, both upper and lower case, a total of 52 variables. You can use both A and a in the same program: the computer will recognise that they are different.

It is wise to make the name you choose as meaningful as possible: for example, you could label a variable representing the *height* of a rocket, H.

LET

LET...= allows you to assign a value to a variable. The value can either be a number or an expression, so in the program above we have

```
LET a=0
```

and

```
LET a=a+1
```

The expressions can be very complex:

```
LET a=EXP(-x**2/2)/SQR(2*PI)
```

You can assign values to more than one variable in a single LET command:

```
LET A=6, b=12, C=25.......
```

The variables must be separated by commas.

You may have noticed that when it is used with LET, an = sign does not mean 'equals' but 'becomes equal to'.

SWAP

SWAP allows you to exchange the values of two variables. Suppose you have two variables a and z and want to swap their values; SWAP has this format:

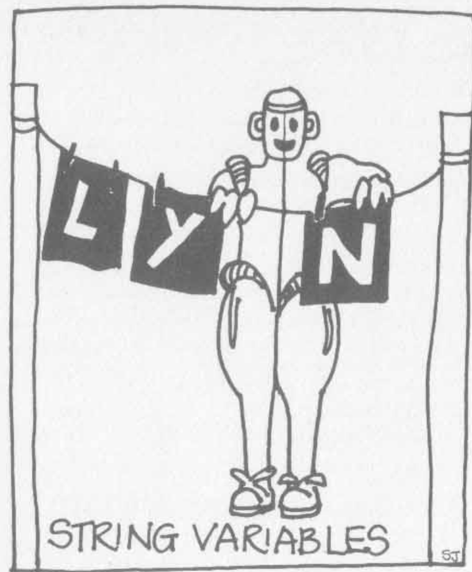
```
SWAP a,z
```

SWAP is especially useful for sorting values in order of size. For example, if you want to sort several numbers into numerical order, you can write a program which compares the size of each pair of numbers in turn, and swaps them if the first is bigger than the second (see Decision making, Chapter 6).

You cannot swap **string variables.**

STRING VARIABLES

As mentioned earlier, a 'string' is a collection of characters. A **string variable** is similar to a numerical variable: it is a **label** referring to a location in memory in which a **string**, rather than a numerical value, is stored. You can use up to 26 string variables in a program. They must be labelled `A$`, `B$`, `C$`, and so on, up to `Z$`. (`$` is the standard symbol for representing strings). Ordinarily, the strings represented by string variables can contain up to 16 characters, including any spaces. (In Chapter 7, we will explore strings in more detail).



You can assign a 'value' to a string variable using `LET`; the value must be placed in inverted commas. For example:

```
LET A$="LYNX"
```

You can use string variables in `PRINT` and `INPUT` statements. Try typing in the following program:

```
10 INPUT "What is your name";A$ [RETURN]
20 PRINT "Hello ";A$ [RETURN]
RUN [RETURN]
```

We will look at `INPUT` in some detail a little later.

You can probably see that by using string variables carefully, you can give your programs a professional look. You can even give the computer a character, and make it seem to show an interest in the person using it (see the example program at the end of this chapter).

You can ask the computer to display the value of any variable at any time by typing in the variable name, `a`, `b`, `c` or whatever, and pressing *[RETURN]*.

The computer stores the values of variables separately from the rest of the program, in a specially constructed table. Every time a variable is used, the computer consults the table. Any values which are changed as a line is executed are updated.

INPUT

`INPUT` tells the computer to expect a response typed in through the keyboard. For example, in the following program, users have to type in their age:

```
10 INPUT A [RETURN]
20 PRINT "Your age is ";A [RETURN]
RUN [RETURN]
```

The computer reads `INPUT`, and waits for a response, printing a `?` to indicate that it is waiting. On the Lynx, your response can be either a number or an **expression**, so if you are, say, 20, you can type in 20, or you could type in 1982-1962.

Once the information has been typed in *and* `RETURN` *has been pressed*, the computer processes it according to the instructions on the following line. In this case it stores it as a variable called `A`, then prints the value of `A` on the screen.

`INPUT` also allows you to combine printing information on the screen with typing in a response to the program. You can include a string in the input command, which will be printed on the screen before the `?`, as part of the prompt. The string's length is limited by the maximum line length of 240 characters.

If the program above is altered so that line 10 reads

```
10 INPUT "What is your age";A [RETURN]
```

then instead of a rather obscure question mark, the computer will indicate that it expects a response by asking

```
What is your age?
```

As in the case of `PRINT`, you need to use a comma or a semi-colon to set the amount of space left between the two components of the line, the string and the variable.

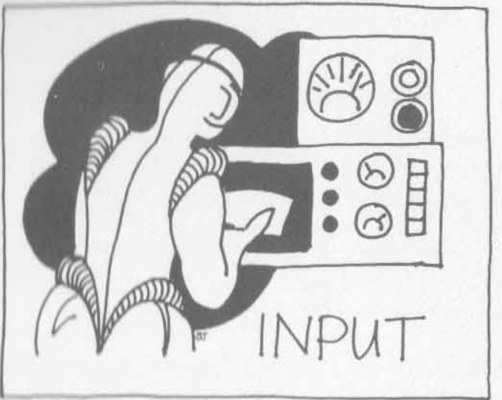
You can arrange an `INPUT` so that the computer expects more than one response. For example,

```
10 INPUT A,B,C
```

In this example, when it comes to line 10, the computer will respond with a `?` prompt. Your responses must be separated by a comma, otherwise the computer will not know that they are three separate values and will treat them as one high value. In this situation, being able to insert an instruction into the input prompt is very useful. For example:

```
10 INPUT "What are the three values A,B,C";A,B,C,
```

will make things much easier for the user.



Remember that INPUT automatically adds a question mark to whatever you include in the string, so you should phrase it accordingly.

It is difficult to stop a program with [ESC] whilst the computer is waiting for an input: this can be useful, because it means that an inexperienced user is unlikely to stop the program accidentally. If you want to use [ESC] during an input, you must first type in an acceptable answer, then hold down the ESC key, then press [RETURN].

One interesting aspect of both PRINT and INPUT is that, with careful wording, you can use them to give the computer a personality.

IDEAS and EXAMPLES

1. This program is a very simple example of giving the computer a personality using PRINT, INPUT and string variables.

```
10 INPUT "Hello, what is your name";A$ [RETURN]
20 PRINT A$; [RETURN]
20 INPUT ", how old are you";a [RETURN]
30 PRINT "Where do you live, "; A$; [RETURN]
40 INPUT B$ [RETURN]
50 PRINT "And have you lived there ";a;" years"; [RETURN]
60 INPUT C$ [RETURN]
70 PRINT "I've never been to ";B$;". What's it like there"; [RETURN]
80 INPUT D$ [RETURN]
90 PRINT "What, is it really ";D$;"?" [RETURN]
RUN [RETURN]
```

You might like to try writing a similar program, using your knowledge of the person you intend to show it to, to make the computer's remarks seem appropriate.

Programming computers to understand – or seem to understand – natural language is a branch of Artificial Intelligence. If this interests you, you might like to read 'Experiments in Artificial Intelligence for Small Computers' by John Krutch (published by Howard W. Sams & Co, Inc).

As you explore Basic more fully, you will be able to write more sophisticated examples.

2. Here is very simple accounting program!
(You can change 'pounds' to 'pence' if you are poor).

```
100 INPUT "How much money (in pounds) do you receive per week";C
[RETURN]
110 INPUT "How much (in pounds) do you spend per week";D [RETURN]
120 CLS [RETURN]
130 PRINT "MONEY RECEIVED";TAB 18;"MONEY SPENT"; TAB 29;"BALANCE"
[RETURN]
140 PRINT "IN POUNDS";TAB 18;"IN POUNDS";TAB 29;"IN POUNDS" [RETURN]
150 PRINT [RETURN]
160 PRINT TAB 15;"PER WEEK:" [RETURN]
170 PAUSE 2500 [RETURN]
180 PRINT [RETURN]
190 PRINT C;TAB 18;D;TAB 29; C-D [RETURN]
200 PRINT [RETURN]
210 PRINT "At this rate, the figures per year will be:" [RETURN]
220 PAUSE 2500 [RETURN]
230 PRINT [RETURN]
240 PRINT C*52;TAB 18;D*52;TAB 29;(C-D)*52 [RETURN]
RUN [RETURN]
```

The PAUSE in line 170 and in line 220 is intended to give the impression that the computer is making some weighty calculation.

Chapter 5: LOOPING

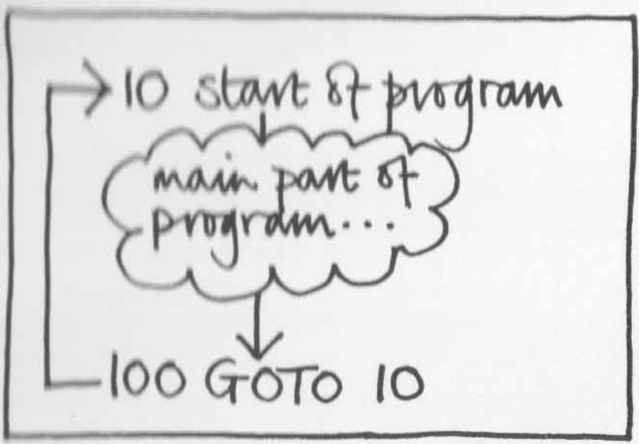
GOTO

As we saw in Chapter 4, GOTO alters the pattern of program execution: instead of executing the program lines in numerical order, the computer, when it reads

```
GOTO line number
```

moves to the line specified, ignoring everything between. The command is sometimes called an 'unconditional jump' because the computer does not have to make a decision before it obeys. (In Chapter 6 we will see the computer considering specified circumstances and deciding whether or not to jump).

GOTO is used to create **loops** within a program.



You can make a program run again and again, without having to enter RUN every time, by including a GOTO command at the end which sends the computer back to the beginning again.

When a program forms a continuous loop it can be stopped by pressing the [ESC] key.

Use GOTO carefully. It is easy to fall into the habit of writing programs which have operations arranged haphazardly and linked together by GOTO, but this is considered bad programming style. If your program is clearly organised it will be easy to modify and improve.

Using everything we have seen so far it is already possible to write useful programs: here, for example, is a short program to calculate the hypotenuse of a triangle:

```
10 INPUT "Lengths of sides A,B,in centimetres ";
A,B [RETURN]
20 LET C= SQR(A**2+B**2) [RETURN]
30 PRINT "The length of side C is ";C;"centimetres
." [RETURN]
40 GOTO 10 [RETURN]
RUN [RETURN]
```

This program will run again and again because the GOTO in line 40 sends the computer back to the beginning of the program. Each time it runs, the result will be printed

below the previous one. If you want to clear the screen between each calculation you can add

```
5 CLS [RETURN]
```

and

```
35 PAUSE 10000 [RETURN]
```

and change line 40 to

```
40 GOTO 5 [RETURN]
```

PAUSE

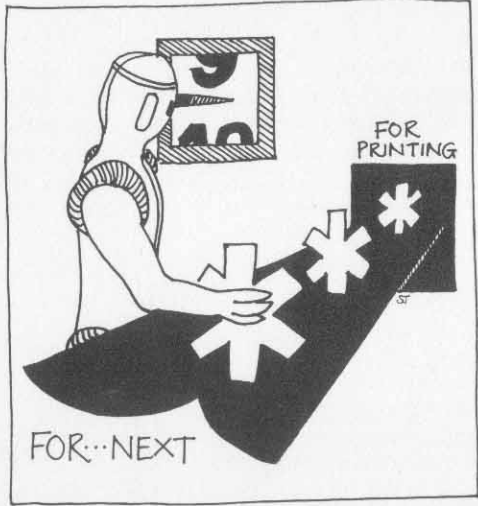
You may find that in a program like the one above, which displays information on the screen and then reruns itself, the information disappears from the screen so quickly that you scarcely have time to read it. `PAUSE` allows you to specify how long the information is to be displayed. It has this format:

```
PAUSE number
```

When it reads `PAUSE` the computer starts up an internal timing loop, and runs it for the number of times specified in the command. The loop lasts for about one ten thousandth of a second, so

```
PAUSE 10000
```

will give you a pause of roughly one second. It is best to experiment with numbers to find the length of pause you need in a particular situation.

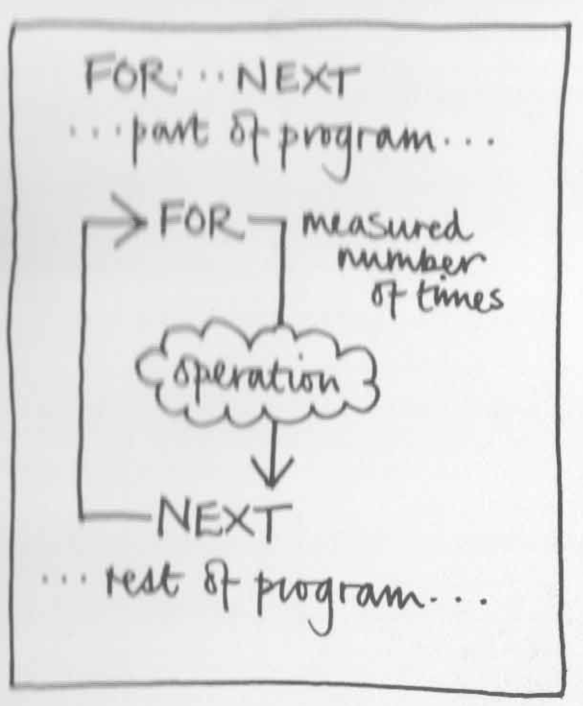


The FOR...NEXT loop

The computer is very good at repeating operations, and as we have seen with `GOTO`, programs can be made to loop. There is another Basic structure, the `FOR...NEXT` loop, which allows you, as part of a bigger program, to repeat an operation for a specific number of times. It looks like this:

```
FOR variable=initial value TO final value
operation
NEXT variable
```


The first line tells the computer to set up a counter in the form of a variable with precise initial and final values. Starting with the variable at its initial value, the computer executes the operation ; when it reaches `NEXT`, it adds one to the value of the counter variable, then executes the operation again, and so on, until the counter variable reaches its final value. The computer then moves on to the rest of the program.



To see this more clearly, try typing in and running this program:

```
10 FOR J=0 to 10 [RETURN]
20 PRINT J;" "; [RETURN]
30 NEXT J [RETURN]
```

The computer will print out

```
0 1 2 3 4 5 6 7 8 9 10
```

as the counter variable, `J`, increases each time the loop is executed.

You can specify the rate of increase, the increment', using `STEP`. Try changing line 10 in the program above to

```
10 FOR J=0 TO 10 STEP 2 [RETURN]
```

This time the computer should print

```
0 2 4 6 8 10
```

You can also make the loop decrease by stepping by a minus number, like this

```
10 FOR J=10 TO 0 STEP-1 [RETURN]
```

which will result in

```
10 9 8 7 6 5 4 3 2 1 0
```

If you do not specify the stepping rate, the counter variable automatically increases by one.

If you want to have another look at your program, type

```
LIST [RETURN]
```

You can see that the lines contained in the `FOR...NEXT` loop are printed indented; this is to make the program's structure clearer.

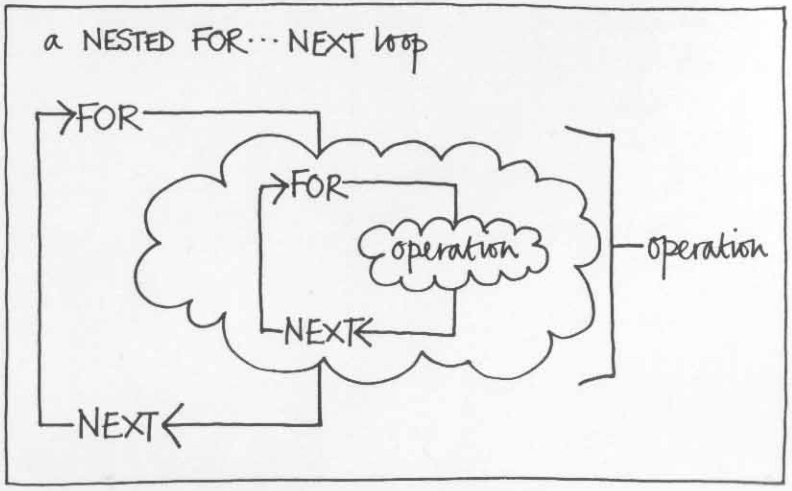
To run the program again, simply type

```
RUN [RETURN]
```

Here's an example of a `FOR...NEXT` loop running from 1 to 10: the Lynx patent dog walk calculator!

```
10 PRINT "Distance"; TAB 20; "Time" [RETURN]
20 FOR D=1 TO 10 [RETURN]
30   LET M=D*15 [RETURN]
40   PRINT D;" miles"; TAB 20;M;" minutes"[RETURN]
50 NEXT D [RETURN]
60 END [RETURN]
RUN [RETURN]
```

`FOR...NEXT` loops can be **nested** – that is, one `FOR...NEXT` loop can run **inside** another.



Look carefully at this program:

```
10 REM  nested FOR...NEXT loops [RETURN]
20 FOR J=0 TO 10 [RETURN]
30   FOR I=0 TO 10 [RETURN]
40        PRINT "*"; [RETURN]
50   NEXT I [RETURN]
60   PRINT [RETURN]
70 NEXT J [RETURN]
RUN [RETURN]
```

Lines 20 and 70 set up a `FOR...NEXT` loop which counts from 0 to 10, so any operation contained in the loop will be repeated 11 times. The operation consists of another `FOR...NEXT` loop, lines 30-50, and a print statement, line 60. The 'inner' `FOR...NEXT` counts from 0 to 10, printing a * each time. The semi-colon following the * tells the computer to continue printing on the same line. At the end of the inner `FOR...NEXT` loop

there is an empty `PRINT` statement which, because it does not end with a semi-colon, brings the cursor down to the next line, ready for the next run.

Try changing line 30 to

```
30 FOR I=0 to J
```

Then the number of *s printed on each line will increase as the counter variable, `J`, of the outer `FOR...NEXT` increases.

END

A program will stop running when the computer has used up all its instructions. But you can also stop it by inserting an `END` statement. This is good programming style: following an `END` statement, the computer will print

```
Ready!
```

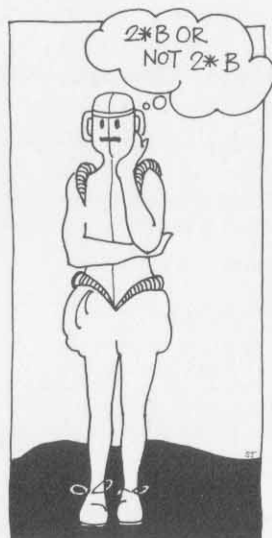
`END` is important in complex programs – see Chapter 6, Making decisions.

IDEAS and EXAMPLES

Here is a program which prints out the three times table, but you can easily change it to print out other tables.

```
100 FOR J=0 TO 12 [RETURN]
110 LET N=J*3 [RETURN]
120 PRINT J;" X 3 = ";N [RETURN]
130 NEXT J [RETURN]
```

Chapter 6: MAKING DECISIONS



IF...THEN

With `IF...THEN` you can ask the computer to make decisions.

`IF` is followed by an expression defining a particular condition, or set of conditions, and instructs the computer to test whether the condition is fulfilled. If it is, the computer will execute the rest of the line, consisting of `THEN` followed by an instruction. If the condition is not fulfilled, the computer ignores the rest of the line and passes to the next line.

CONDITIONS

So far we have talked about conditions being 'fulfilled', or 'not fulfilled', but these are imprecise terms, allowing shades of meaning. The computer can only decide between two possibilities: the condition can be either *true* or *false*: there can be no shades of meaning. To program the computer to make complex decisions you first need to break the decision down into a series of true- or false-type conditions.

Let's see how this works in an example, using the simplest of conditions – 'equality', represented by an `=` sign. In this context, the `=` strictly means 'equal to' (not 'becomes equal to', as it did with `LET` – see Chapter 4).

```
10 LET A=INT(RND*10)+1 [RETURN]
20 PRINT "I'm choosing a number...." [RETURN]
30 INPUT "What is my number, between 1 and 10";B [RETURN]
30 IF A=B THEN PRINT "You were right!" [RETURN]
RUN [RETURN]
```

The decision is made in line 30: if `A` equals `B`, the computer responds by printing `You were right!`; if they are not equal, it does nothing.

The truth or falsehood of a condition is called its **logical state**. ('Logical', in this context, simply means that the state is confined to one of two possibilities, true or false). Depending upon its logical state, the computer assigns to the condition a **logical value** of either 0 or 1: 0 if it is false, 1 if it is true. So, in the context of an `IF...THEN` command, if the expression following the `IF` has a value of 0, the computer ignores the `THEN...`; if it has a value of 1, the computer executes the `THEN...`.

The OPERATORS

The conditions always involve *comparing* values in one way or another. The symbols which represent the different types of comparisons are called **operators**, the things compared are called **operands**.

'Equals' is only one example of a group of operators called **relational operators**. These include

`<` 'less than'
`>` 'greater than'
`<=` 'less than or equal to'
`>=` 'greater than or equal to'
`<>` 'not equal to'

These operators are all used with `IF...THEN` to form conditions.

The conditions can be **combined**, then tested and assigned a logical value of 0 or 1, using the three **logical operators**

```
AND
OR
NOT
```

(which must be followed by a space). These operators ask the computer to combine conditions in different ways.

Let's look at each one in turn. Suppose we take two conditions: they could be anything, but we will use

```
A<=5
B>100
```

AND

```
IF A<=5 AND B>100 THEN PRINT "*"
```

The `IF...` part of this will be assigned a value of 1 (TRUE) only if both the `A<=5` *and* `B>100` are true; then the computer will print a *. Otherwise, if one or other, or both are false, the whole construction is assigned a value of 0 (FALSE) and nothing is printed.

OR

```
IF A<=5 OR B>100 THEN PRINT "*"
```

Here the `IF...` will be given a value of 1 (TRUE) if either `A<=5` *or* `B>100` is true, or *if both are true*, but a value of 0 (FALSE) if *both* are false.

NOT

`NOT` is used less often than `AND` and `OR`: it returns a value of 1 (TRUE) if the condition it tests is FALSE, and a value of 0 (FALSE) if the condition is TRUE. `NOT` is at its most useful with **strings**, or in long, complicated combinations of conditions.

In addition, the computer can recognise a sort of 'implied' operator, and assign a logical value to a variable depending on whether or not its numerical value is zero: it is given a value of 0 if it is zero, 1 if not. So

```
IF A THEN PRINT "*"
```

will result in a star being printed if `A` is not 0.

```
IF NOT A THEN PRINT "*"
```

will print a star if `A` is 0.

THE HIERARCHY OF OPERATIONS

When performing these comparisons, the computer follows a strict order of operations:

any arithmetical operations are carried out first;

then the relational operators, =, <, >, <=, >=, <> are processed, in order from left to right;

then NOT;

then AND;

and finally, OR.

You can alter this order by inserting brackets: any operation in brackets will be processed first.

The OPERANDS

Operators can be used to compare numbers, variables, functions, and in some cases strings. Let's look at some examples:

```
IF A=B THEN PRINT "*"
```

A star will be printed only if A has the same value as B.

```
IF A=SIN(B) THEN PRINT "*"
```

A star will be printed only if A has the same value as SIN(B).

IF...THEN with STRINGS

Strings and string expressions can be compared only with the = operator. For example:

```
IF A$="LYNX" THEN PRINT "*"
IF A$=B$ THEN PRINT "*"
IF A$="LYNX" + B$ THEN PRINT "*"
IF LEFT$(A$,3)= C$ + CHR$ (84) THEN PRINT "*"
```

You can also construct the equivalent of <>, using NOT like this:

```
IF NOT A$=B$ THEN PRINT "*"
```

Almost any command can be made conditional using IF...THEN. If it follows THEN, a command has the same format as it does normally.

IF...THEN...ELSE

Unless the THEN part of an IF...THEN directs it elsewhere, the computer will always execute the line that follows, whether it executed the THEN... or not.

If you want to create a **branch** in your program, and have the computer execute *one of two alternative possibilities*, you can use IF...THEN...ELSE.

It is used like this:

```
IF condition THEN operation
ELSE alternative operation
```

If the condition is true, the computer executes the operation following THEN, then skips over the next line when it reads ELSE. If it has not executed the THEN..., it executes the ELSE. Look at this program:

```
10 INPUT "What is your name"; A$
20 IF A$="LYNX" THEN PRINT "That's my name too!"
30 ELSE PRINT "That's a funny name!"
```

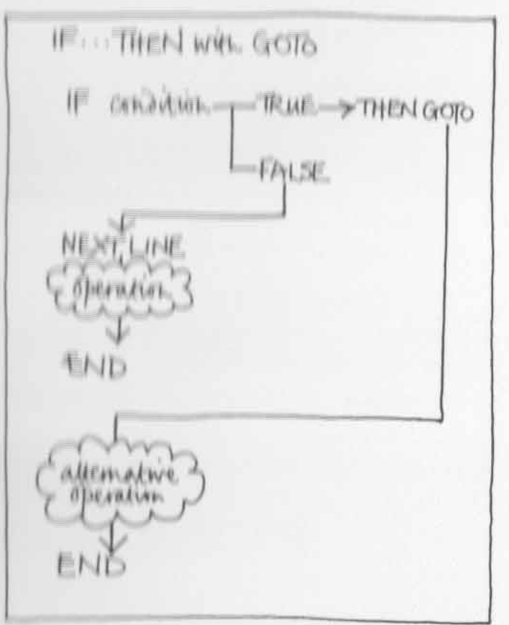
The decision is made in line 20: if your response is LYNX, the computer prints That's my name too!, but not That's a funny name!. Otherwise, it prints That's a funny name!.

Note that the alternatives can only be *single commands*.

(If, by mistake, you type in an ELSE line without a FOR...NEXT, the computer will ignore it but will not display an error message).

IF...THEN with GOTO

If the THEN... part, or the ELSE... part, of your decision is longer than a single command, you can use IF...THEN with GOTO to direct the computer to operations in other parts of the program.



Programs using IF...THEN with GOTO are usually very long, but here is a rather artificial short example. At the end of the section you will find a program which simulates tossing a coin, using IF...THEN ELSE. Here is how you might write it if IF...THEN ELSE did not exist:

```
5 PAUSE 10000 [RETURN]
10 PRINT "I'm tossing the coin..."; [RETURN]
20 FOR J=1 TO 3 [RETURN]
30   PAUSE 5000 [RETURN]
40   PRINT "."; [RETURN]
50 NEXT J [RETURN]
60 PRINT [RETURN]
70 IF RND<0.5 THEN GOTO 100 [RETURN]
80 PRINT "and it's TAILS!" [RETURN]
90 GOTO 5 [RETURN]
100 PRINT "and its HEADS!" [RETURN]
110 GOTO 5 [RETURN]
RUN [RETURN]
```

The first possible operation is contained in line 100, the second in line 80. If it prints TAILS, the computer is prevented from printing HEADS as well by the GOTO 5 in line 90.

The two alternatives could, of course, be much longer than one line.

On the Lynx, the line number following `GOTO` can be represented by a variable or an expression; so you can have

```
GOTO A
GOTO 100+INT(RND*6)
```

and so on.

This means that you can also use `GOTO` to make very complex branches in your programs – provided you are very careful!

```
100 DIM A$(30) [RETURN]
110 INPUT "Hello, what's your name";A$ [RETURN]
120 GOTO 120+RAND(4)+1 [RETURN]
121 PRINT "That's a nice name, ";A$ [RETURN]
122 PRINT "That's a funny name-";A$;"!" [RETURN]
123 PRINT "Pleased to meet you, ";A$ [RETURN]
124 PRINT "Hello ";A$;", my name is Lynx" [RETURN]
RUN [RETURN]
```

IDEAS and EXAMPLES

Here is a program which simulates tossing a coin, using `IF...THEN ELSE`.

```
10 PRINT "I'm tossing the coin..."; [RETURN]
15 FOR J=1 TO 3 [RETURN]
20   PAUSE 5000 [RETURN]
25   PRINT "."; [RETURN]
30 NEXT J [RETURN]
40 PRINT [RETURN]
50 PRINT "and it's "; [RETURN]
60 IF RND< 0.5 THEN PRINT "TAILS!" [RETURN]
70 ELSE PRINT "HEADS!" [RETURN]
80 PAUSE 10000 [RETURN]
90 GOTO 10 [RETURN]
RUN [RETURN]
```


Chapter 7: MORE ABOUT STRINGS

We have already used strings with `PRINT` and `INPUT`, and string variables like `A$` and `B$`. In this section we will explore other ways of using strings.

We have seen that you can assign a value to a string variable using `LET`,

```
LET A$="LYNX"
```

and that, ordinarily, the computer will allow you to type in a string of up to 16 characters. In fact, you can tell the computer exactly how long you want the string to be (**dimension** it), up to a maximum of 127 characters, using `DIM`.

DIM

```
DIM A$(6)
```

tells the computer to accept a string up to 6 characters long. If you type in more than 6 characters, the computer will **truncate** your string – it will ignore the excess characters.

Try typing in this program:

```
10 DIM A$(8) [RETURN]
20 INPUT "A$";A$ [RETURN]
30 PRINT A$ [RETURN]
40 GOTO 20 [RETURN]
RUN [RETURN]
```

Try entering strings of different lengths and see what the computer does to them.

The `GOTO` in line 40 sends the computer back to line 20, not line 10, because the string does not need to be re-dimensioned. If it was re-dimensioned, the computer would use another chunk of memory space for storing it, without erasing the earlier area, and eventually run out of memory.

CHR$

Each of the characters that can be displayed on the screen has a special code number, which allows the computer to identify it. The Lynx uses a standard set of codes, called ASCII (American Standard Code for Information Interchange). You can find a list of the ASCII codes in Appendix 3.

Using

```
CHR$ (code number)
```

you can tell the computer to convert a code number into the character it represents. The code number may be represented by an expression, like

```
CHR$(A)
```

or

```
CHR$(A*10)
```

`CHR$` can be used in `PRINT` statements, or in **string expressions** – see later in this chapter.

Here's a program using `CHR$` which displays the character set:

```
100 FOR J=32 TO 127 [RETURN]
110 PRINT CHR$(J) [RETURN]
120 NEXT J [RETURN]
RUN [RETURN]
```

KEYN and KEY$, GETN and GET$

In calculator mode you can find the ASCII code of a character by typing `GETN` *[RETURN]*, then typing in the character. The computer will display the ASCII number.

`KEYN` and `GETN` both return the ASCII code of the key currently pressed; if no key is pressed, `KEYN` will give 0, but `GETN` will *wait until a key is depressed*, then return its code.

`KEY$` and `GET$` are similar, but they return the **character string** of the key pressed. If no key is pressed, `KEY$` will return a 'null string' (nothing), `GET$`, like `GETN`, will wait.

These functions (like `RAND`) can be used to assign values to variables

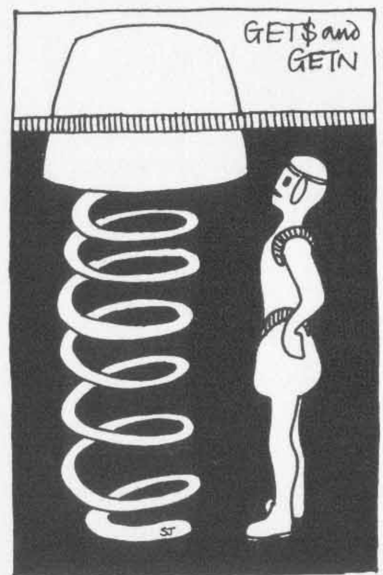
```
LET V=KEYN
LET V$=KEY$
```

and with `IF...THEN`.

If, for example, you wanted to move a token around the screen using the arrow keys you might do it like this:

```
IF KEYN=123 THEN LET C=C+1
IF KEYN=124 THEN LET C=C-1
```

and so on, where the token's position on the screen is represented by co-ordinates `C, L` (column, line) and `123` and `124` are the ASCII codes of the right and left arrows respectively.



PARTS OF STRINGS

There are various commands which allow you to select *parts* of strings; these are especially useful with `IF...THEN`.

LEFT$ and RIGHT$

With `LEFT$` you can select the left-hand side of a particular string, like this:

```
LEFT$(string name, number of characters wanted)
```


If `A$="LYNX"`, then

```
LEFT$(A$,2)
```

will give you `LY`.

Don't forget the brackets or the commas! The number of characters can be represented by an expression.

`RIGHT$` is similar, but gives the right-hand side of the string, so

```
RIGHT$(A$,2)
```

will give you `NX`.

MID$

`MID$` allows you to select any part of a string by specifying the number of characters you want starting from a particular point in the string, like this:

```
MID$(string name, number of first character wanted,
                  number of characters)
```

LYNX has four characters: L is number 1, Y number 2, and so on. If you wanted `YN`, you would use `MID$` like this (`A$="LYNX"`):

```
MID$(A$,2,2)
```

Remember that the numbers can be represented by expressions.

VAL

`VAL` allows you to select the numerical part of a string and process it like an ordinary number, provided it is at the *beginning* of the string. If `A$` is `32 pounds`, then

```
VAL(A$)
```

will give you `32`.

If there is no number in the string involved, the computer will return a value of `0`.

`VAL` can select **hexadecimal** numbers, provided they are marked with an `&` (for hexadecimal numbers, see Chapters 15 and 16 on machine code).

ASC

`ASC(A$)`

will give you the code number of the first character of `A$`. So if `A$=LYNX` it will return a value of `76`.

If `A$=""` (a null string), `ASC(A$)` will give you `13`, which is the code number for a carriage return (see Chapter 14).

UPC$

`UPC$(A$)`

will convert all letters in `A$` to upper case (but leave any numbers or symbols unaltered).

LEN

`LEN(stringname)`

gives you the length of the particular string, that is, the number of characters in it. So, if `A$="LYNX"`,

```
LEN(A$)
```

will be `4`.

STRING EXPRESSIONS

You can join strings together, **concatenate** them, like this:

```
10 LET A$= "LYNX" + " COMPUTER" [RETURN]
20 PRINT A$ [RETURN]
RUN [RETURN]
```

and this:

```
100 CLS [RETURN]
110 LET A$="32" [RETURN]
120 LET B$="110" [RETURN]
130 PRINT "A$=";A$ [RETURN]
140 PRINT "B$=";B$ [RETURN]
150 PRINT [RETURN]
160 PRINT "A$+B$="; A$+B$ [RETURN]
170 PRINT [RETURN]
180 PRINT"VAL(A$)+VAL(B$)=";VAL(A$)+VAL(B$) [RETURN]
190 END [RETURN]
RUN [RETURN]
```

IDEAS and EXAMPLES

Here is an example of string-handling:

```
100 REM ANIMAL [RETURN]
110 CLS [RETURN]
120 DIM A$(24) [RETURN]
130 LET A$="DOGCATANIMALLYNXCOMPUTER" [RETURN]
140 PRINT "A$ is ";A$ [RETURN]
150 PRINT [RETURN]
160 PRINT "A$ has ";LEN(A$);" characters" [RETURN]
170 PRINT "The rightmost 8 characters are "; [RETURN]
180 PRINT RIGHT$(A$,8) [RETURN]
190 PRINT "The leftmost 3 characters are "; [RETURN]
200 PRINT LEFT$(A$,3) [RETURN]
210 PRINT [RETURN]
220 PRINT "A ";RIGHT$(A$,8);" is a ";MID$(A$,13,4) [RETURN]
230 PAUSE 10000 [RETURN]
240 PRINT "A ";MID$(A$,13,4);" is a ";MID$(A$,4,3) [RETURN]
250 PAUSE 10000 [RETURN]
260 PRINT "A ";MID$(A$,4,3);" is an ";MID$(A$,7,6) [RETURN]
270 PAUSE 10000 [RETURN]
280 PRINT [RETURN]
290 PRINT "Therefore "; [RETURN]
300 PAUSE 10000 [RETURN]
310 PRINT "a ";RIGHT$(A$,8);" is an ";MID$(A$,7,6) [RETURN]
RUN [RETURN]
```


Chapter 8: EDITING

In this chapter we will look at the facilities for editing programs.

Sometimes you will want to alter your programs. You may want to improve a program, or make it do something different. Or you may have made an error you need to correct. The error may be something simple which does not stop the program running, a mistyped string for example. Or your program may have a **bug**.

A bug is an error in a program. We have already seen that the computer checks each line as it is typed in, and that if a line does not make sense to the computer it will respond with an error message. This kind of error is usually a **syntax error**, which means that the line does not obey the rules of the Basic language; perhaps a command has been misspelt, or even omitted.

But a bug is a different type of error: something which is acceptable to the computer, but which makes your program do things you did not intend. A bug may be the result of a typing mistake. But it may be a fault in the construction of your program. Finding and removing bugs is called **debugging**.

REM

The REM command allows you to insert comments (REMarks!) into a program. These are ignored by the computer whilst it is executing the program, but are saved and listed like ordinary lines.

```
10 REM comment [RETURN]
```

A REM may be inserted at any point in the program, and is used to label its various parts. In a game, for example, you might have

```
10 REM Setting up the board
100 REM Moving the players
200 REM The score
```

You can even have comments like

```
600 REM The program works up to here.
```

In fact, REM is valuable tool: if each part of a program is labelled it is easier to correct or improve it; it is also easier for other people to understand it.

LIST

Before you can edit a program, of course, you need to be able to examine it. LIST [RETURN] tells the computer to print a list of your program on the screen. When it reaches the bottom of the screen, the computer will start writing at the top again, so to halt the listing for a time, press the [SHIFT] key; to start it again, release the [SHIFT] key.

You can also ask the computer to list particular parts of the program. For example:

```
LIST 100 [RETURN]
```

will list line 100 only. You can also specify blocks of program for listing:

```
LIST 100,200 [RETURN]
```

will list all lines numbered from 100 to 200 inclusive.

DEL

You may find that you need to remove large quantities of the program. DEL allows you to delete individual lines, and blocks of lines, leaving the rest of the program unaltered. It has a similar format to LIST. You can, for example,

```
DEL 100 [RETURN]
```

which will delete line 100, or you can

```
DEL 100,200 [RETURN]
```

which will delete all the lines from 100 to 200 inclusive.

ESCAPE and CONT

A program may run for a long time, or may form a continuous loop which would run forever, and it may not be doing what you expected it to do. By pressing the [ESC] key you can stop the program during execution. The computer will display on the screen

```
Stopped in line....
```

telling you which line was being executed when the key was pressed. The program is not impaired by this interruption, and can be restarted by either CONT or RUN.

CONT [RETURN] restarts the program from the point where it was stopped. RUN [RETURN] restarts it from the beginning. These must be used in different contexts.

Whilst the program is stopped, the computer can be used in immediate mode, for listing, examining or altering the values of variables, and so on, or for calculations which have nothing to do with the program at all. If these do not alter the structure of the program, you can use the CONT command to restart the program. If, however, you save or edit the program, you will have to restart it by using RUN.

If you try to use CONT in the wrong situation, the computer will print Cannot continue. If this happens, the program will be intact, and can be restarted with RUN.

(If you are losing a game, you can also, of course, cheat by escaping from the program, then restarting it from the beginning using RUN!)

STOP

STOP is similar to [ESC]. It stops the computer during execution of a program – but unlike [ESC], it is written into the program. For example:

```
100 STOP [RETURN]
```

As with [ESC], the computer tells you which line was being executed when the STOP command appeared; you can use the computer in immediate mode, and restart the program using CONT or RUN, whichever applies.

STOP is primarily a debugging tool. It can be inserted at various points in a program allowing you to try out small sections, monitor the values of variables, make small alterations, and so on. Once your program is running correctly, you can delete the STOP commands.

TRACE and SPEED

Two other debugging aids are TRACE and SPEED.

```
TRACE ON [RETURN]
```

tells the computer to display the number of the line it is about to execute. If your program is doing something you did not expect, you can turn on TRACE, run the program and track down the line which is at fault. To turn TRACE off, type

```
TRACE OFF [RETURN]
```

TRACE ON/OFF can be used inside a program, so you can use it on particular sections of program.

FOR... and REPEAT will only be displayed the first time they are executed; ELSE is viewed as an extension of the IF...THEN line above, and is not displayed.

TRACE is normally off.

Using

```
SPEED number between 1 and 255
```

you can slow your program down, so that you can see exactly what it is doing. It works by increasing the delay between program lines.

SPEED 1 is fastest,

SPEED 255 is slowest,

SPEED 0 returns to normal.

Like TRACE, SPEED can be used inside a program, so you can use it on particular sections.

RENUM

RENUM allows you to **renumber** your program. You use it like this:

```
RENUM number of first line, rate of increase
```

so

```
RENUM 1000,100
```

would renumber your program so that its first line was 1000, and each subsequent line number increased by 100. Alternatively you can use

```
RENUM number of first line
```

in which case the computer will increase each line number by 10.

If you specify neither, the computer will start renumbering at 100 and increment by 10.

The numbers following RUN (Chapter 4), GOTO (Chapter 5), RESTORE (Chapter 10), and GOSUB (Chapter 11), will be altered; but not those following LCTN (see Chapter 16).

If your program contains a GOTO (for example) to a line which does not exist, RENUM will round it up to the next line number.

NEW

If you want to clear your program from the computer's memory, NEW [RETURN] will erase it completely; there is no way of retrieving it. So, be careful with NEW!

EDITING INDIVIDUAL LINES

You will often want to alter a small part of an individual line. Editing on the Lynx has been designed to be quick and easy.

If you have just entered a line, and the computer has displayed a syntax error or similar message, you can enter edit mode by typing

    [CONTROL] Q [RETURN]

and the computer will display the line, with the cursor positioned at the beginning.

If the line you want to edit was entered earlier, type

    [CONTROL] E [RETURN]

The computer will ask

`Line number?`

You then enter the appropriate line number, press [RETURN], and the computer will print the line on the screen, with the cursor at the beginning.

You can move the cursor

to the *left* using the *left arrow*,

to the *right* using the *right arrow*,

to the *beginning of the program line* using the *up arrow*,

to the *end of the program line* using the *down arrow*.

You can insert characters by simply typing them in: any characters to the right of the cursor will be moved along.

Characters to the left of the cursor can be deleted by pressing the [DELETE] key. The characters to the right of the cursor will move back to fill the gap.

When you have finished editing, press [RETURN] – the cursor does not need to be at the end of the line – and the new version of your line will be stored in the computer's memory, replacing the previous version.

Chapter 9: STORING AND LOADING PROGRAMS

You may have noticed that typing programs into the computer can be a long and laborious task. And when you switch the computer off, any program stored in its memory is erased. It is possible, however, to store programs on magnetic material and reload them into memory. Microcomputers use two main types of magnetic material: cassette tapes and floppy disks. Floppy disks have certain advantages over cassettes and are, in particular, much faster to use; but cassettes are much cheaper and more generally available, and for most purposes very efficient.

This chapter describes the commands available for saving, loading and manipulating programs on cassette tape. To do this succesfully it is best to use high quality audio tapes, or special computer tapes, and to take care that you have installed the cassette player correctly (see Chapter 1).

Remember that most cassette tapes have a **leader**, and you will need to wind the tape past this before you can record your program.

To guard against total disaster, it is probably best to make notes as you are writing your program; then if you do make a mistake when trying to save it, you will at least be able to type it in again.

SAVE

To SAVE a program you must first decide on a name for it. The name can be any combination of characters, as many as you like (within the maximum line length of 80 characters). You can save it by typing in:

    SAVE "name" [RETURN]

The name must be in inverted commas.

If there is a remote control facility on your cassette player it will be controlled by the computer: press down the record and play keys – the player will not start until you press [RETURN]. When the recording is complete the computer will stop the cassette player automatically, but make sure you press the stop key on the player as well.

If there is no remote facility on your cassette player, type SAVE, press down the record and play keys, then press [RETURN]. When recording is complete, the computer will display the > prompt on the screen, and you can switch the cassette player off.

Depending on the length of the program, saving can take from a few seconds to several minutes.

You can save a program so that it will run automatically as soon as it is loaded by adding a line number to the end of your SAVE command, like this:

    SAVE "name",10 [RETURN]

There must be a comma between the program name and the line number. The program will then run automatically from line 10.

It is wise to make several copies of your program as recordings can easily be damaged or accidentally erased.

VERIFY

If you want to check that your program has been saved correctly, you can use VERIFY. Rewind the tape. Type

    VERIFY "name"

Press the play key on the cassette player, then press [RETURN]. The computer will read through the program and check that it has not been distorted during saving. If anything

has gone wrong, it will display a Bad Tape message on the screen. Otherwise, it will just display the > prompt.

LOAD

To load a program from cassette when you have a remote facility on your player, press the play key on the cassette player and type

```
LOAD "name"
```

then press the *[RETURN]*. As it reads through the tape, searching for the program you have named, the computer will display on the screen the name of each program it finds. When the loading is complete it will display the prompt on the screen and you can execute the program using RUN.

The cassette player will be stopped by the computer, but you must press the stop key as well.

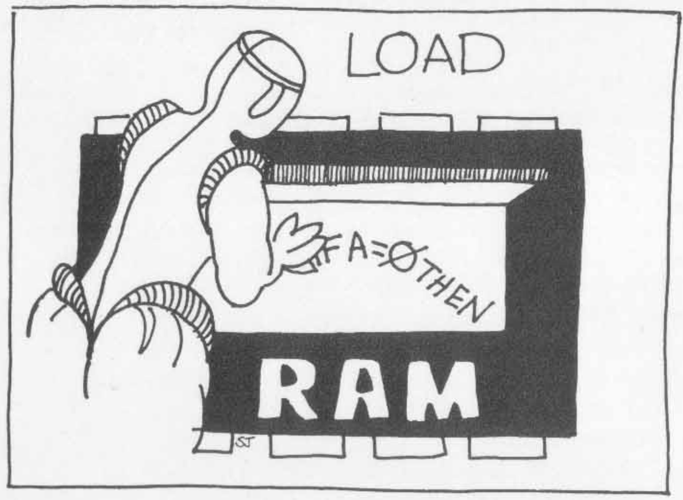
If you have no remote, type

```
LOAD "name"
```

Press the play key, then press *[RETURN]*. When the computer displays the prompt, switch the player off.

If you included a line number in the SAVE command, load the program in the normal way: once it has loaded it will run automatically.

If there was a program stored in the computer's memory before you loaded the program, this will have been replaced by the new program.



APPEND

Unlike LOAD, APPEND allows you to *add* material stored on cassette to the end of a program already stored in the computer's memory. To use it, type

```
APPEND "name"
```

Press the play key, then *[RETURN]*. The first line number of the material you are adding *must* be higher than the last line number of the program already in memory.

You can use APPEND to load programs that have been stored to run automatically.

APPEND is particularly useful for adding **subroutines** to a program. A subroutine is a

distinct part of a program which performs a particular operation – we will explore them in detail in Chapter 11. You can store a 'library' of commonly used subroutines, and append them to programs whenever you need them. But you must remember to store them with high line numbers.

MLOAD

If you want to load a machine code program from tape you must use MLOAD. It is used exactly like LOAD, that is:

```
MLOAD "name"
```

TAPE

Normally, the computer is programmed to SAVE and LOAD programs at a fairly slow baud rate. (Baud is the unit in which 'data flow' is measured). The slower the baud rate, the more tape is used and the more information is recorded, so a slow rate is very reliable.

However, if you are loading and saving very long programs (and want to save time), you can alter the baud rate, using TAPE, like this

```
TAPE number between 0 and 5
```

0 sets the rate at 600 baud (the normal rate)
1 at 900
2 at 1200
3 at 1500
4 at 1800
5 at 2100

To use a high tape speed (4 and 5) successfully, however, you will need a good quality cassette player and good quality tapes.

Chapter 10: MORE VARIABLES

ARRAYS

Suppose you are writing a program which involves processing a large amount of data, and that each item is similar to the others: the items form a 'set'. They could be the examination results of a class of 13 children, for example, which you want to process in some way. You could store each mark as a variable, and process each variable in turn. Or you could set up an array.

An array is a special kind of variable; it consists of **an ordered list of values**. The values have the same variable name, but are differentiated and ordered by **subscripts**, like this:

```
A(0) A(1) A(2) A(3).......
```

Array names are single characters, the letters of the alphabet, both upper and lower case. This means you can have a maximum of 52 arrays in any one program. The subscripts may be numbers, or variables, or expressions. You can have up to 2000 members in an array!

You can have `A$ A a A(x)` and `a(x)` all in the same program, and the computer will recognise that they are all different.

The computer treats all the members of an array as a single entity, and processes each member in turn, at a single command.

DIM

You have to define the size of an array before you use it, so that the computer can set aside memory for storing and manipulating it once the program is running. To do this you use `DIM`:

```
DIM array name(number)
```

where the number is the highest subscript you want. This specifies the size of the array: the computer begins numbering at 0 and continues until it reaches the highest value. So,

```
10 DIM A(12)
```

would set up a list of 13 variables, with subscripts beginning at 0 and ending at 12:

```
A(0) A(1) A(2) A(3)..up to...A(12)
```

So, going back to the class of 13 children, you could write a program to calculate their average mark, and the highest mark, like this:

```
5 REM find the average mark [RETURN]
10 DIM M(12) [RETURN]
20 FOR J=0 TO 12 [RETURN]
30    INPUT "MARK";M(J) [RETURN]
40 NEXT J [RETURN]
50 A=0 [RETURN]
60 FOR J=0 TO 12 [RETURN]
70    A=A+A(J) [RETURN]
80 NEXT J [RETURN]
90 PRINT "The average mark was ";A/13 [RETURN]
100 REM find the highest mark [RETURN]
110 LET H=A(0) [RETURN]
120 FOR J=0 TO 12 [RETURN]
130    IF A(J)>H THEN LET H=A(J) [RETURN]
140 NEXT J [RETURN]
150 PRINT "The highest mark was ";H [RETURN]
RUN [RETURN]
```


You can set up several arrays in a single `DIM` statement, if you separate them with commas:

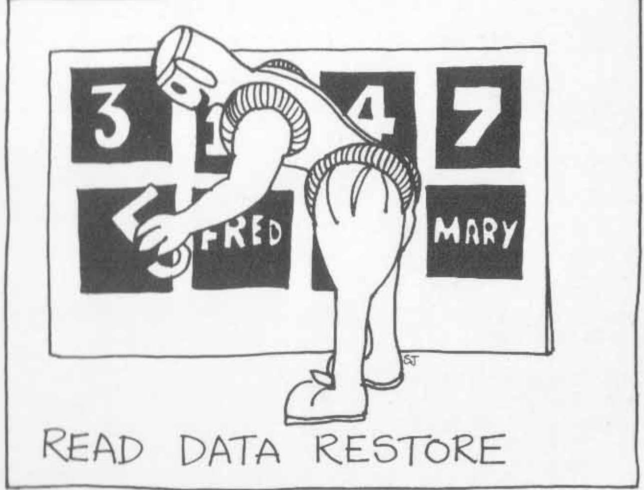
```
DIM A(12), b(5), Z(10)
```

Arrays can have variables or expressions as subscripts. These are dimensioned like this:

```
DIM a(j), B(20*a)
```

`FOR...NEXT` loops are particularly useful with arrays: you can set up a `FOR...NEXT` loop which processes each member of an array in turn. For example, if you want to assign a random value to each member of the array, you can do it like this:

```
10 DIM A(10) [RETURN]
20 FOR J=0 TO 10 [RETURN]
30 LET A(J)=RND [RETURN]
40 NEXT J [RETURN]
RUN [RETURN]
```



READ, DATA....RESTORE

Try this:

```
10 READ A$,B$,C$,D$ [RETURN]
20 PRINT A$;" ";B$;" ";C$;" ";D$ [RETURN]
30 DATA I,am,a,LYNX [RETURN]
RUN [RETURN]
```

`READ`, `DATA` and `RESTORE` are used together. `DATA` allows you to store a series of values (to be assigned to variables) on a program line. The line will be ignored by the computer until it is told by a `READ` command to assign the values to the variables contained in the `READ` statement. `DATA` has this format:

```
100 DATA 100,FRED,e*24,......
```

The values can be numbers or expressions containing variables, or string variables. String variables should not be placed in inverted commas in a `DATA` statement. The data must be separated by commas.

`READ` has this format:

```
100 READ variable1, variable2,.....
```

The variables must be separated by commas, and can be numeric variables, arrays or string variables.
The type of data must match the type of variable: if you tell the computer to

```
READ A
```

and the data it finds is a string, it will display an error message. Similarly, if you ask it to read data and there is none, it will display an `Out of data` message.

When it comes to a `READ` command, the computer assigns to the variables in it the values contained in the data statement, in the order that they appear. As it does so, it keeps track of its position in the data statement using a **data pointer**. If your program contains several `READ` statements you can build up a data block by grouping the `DATA` lines together. By using the data pointer, the computer will know which position in the data block to read from as it executes each `READ` command.

RESTORE

The data pointer can be returned to the beginning of the data block using `RESTORE`, so the data can be used again and again.

It can also be restored to a specified line number within the data block, if only parts of the data are needed. The line number can be specified as a number or as an expression.

Here is an example of how arrays and `READ`, `DATA` might be used together to allow fairly complex processing of information:

```
100 REM DAYS FROM 1st JANUARY [RETURN]
105 DIM N(12) [RETURN]
106 READ N(1) [RETURN]
110 FOR J=0 TO 12 [RETURN]
120  READ N [RETURN]
121  LET N(J)=N+N(J-1) [RETURN]
130 NEXT J [RETURN]
140 INPUT "ENTER DATE dd/mm/yy";D$ [RETURN]
150 LET E$=MID$(D$,2) [RETURN]
160 LET D=VAL(D$)+N(VAL(E$)) [RETURN]
170 LET E$=RIGHT$(D$,2) [RETURN]
180 LET E=VAL(E$) [RETURN]
190 IF D>59 AND FRAC(E/4)=0 THEN LET D=D+1 [RETURN]
200 PRINT "There are ";D;" days from 1st January to ";D$ [RETURN]
210 GOTO 140 [RETURN]
220 DATA 0,31,28,31,30,31,30,31,31,30,31,30 [RETURN]

RUN [RETURN]
```

IDEAS and EXAMPLES

1. This program has a novelty: try running it.

```
100 DIM A(10) [RETURN]
110 FOR J=0 TO 10 [RETURN]
120 READ A(J) [RETURN]
130 NEXT J [RETURN]
140 FOR J=0 TO 10 [RETURN]
150 PRINT CHR$ (A(J)); [RETURN]
160 NEXT J [RETURN]
170 PRINT [RETURN]
180 DATA 73,32,97,109,32,97 [RETURN]
190 DATA 32,76,89,78,88 [RETURN]

RUN [RETURN]
```


2. This program simulates a die. It is an example of restoring to a line number represented by an expression:

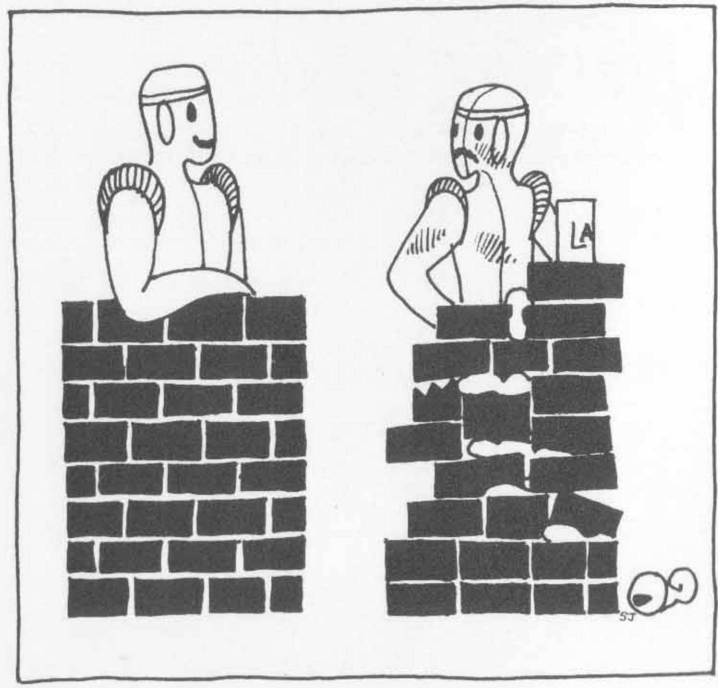
```
10 RESTORE INT(RND*6)*10+1000 [RETURN]
20 FOR J=0 TO 2 [RETURN]
30 READ A$ [RETURN]
40 PRINT A$ [RETURN]
50 NEXT J [RETURN]
60 PRINT [RETURN]
70 PAUSE 10000 [RETURN]
80 GOTO 10 [RETURN]
1000 DATA ...,.*.,... [RETURN]
1010 DATA ..*,...,*.. [RETURN]
1020 DATA ..*,.*.,*.. [RETURN]
1030 DATA *.*,...,*.* [RETURN]
1040 DATA *.*,.*.,*.* [RETURN]
1050 DATA *.*,*.*,*.* [RETURN]

RUN [RETURN]
```

Chapter 11: STRUCTURING COMPLEX PROGRAMS

A program begins as a problem. You organise it, and shape it into a form which allows the computer to solve it. As the the problems you tackle become more and more complex, so your programs will become more and more complex.

But ideally, the shape of your program should always be as clean and simple as you can make it. Fortunately, there is a structure in Basic which allows you to keep complexity under control: the **subroutine**.

A complicated operation, or series of operations, can be made into a subroutine which can be used many times during a program run, and may be used by several different parts of the main program.



GOSUB...RETURN

Subroutines are controlled by two Basic commands, GOSUB and RETURN.

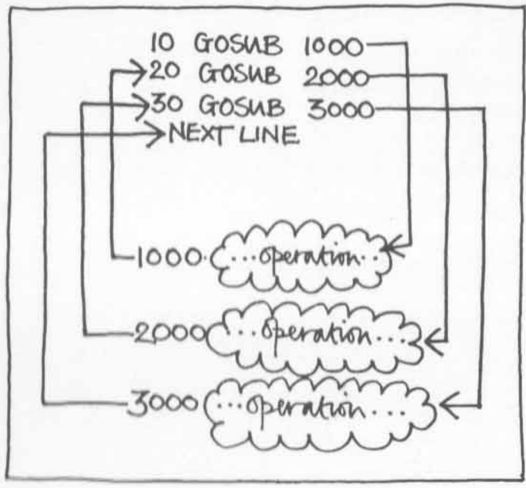
The computer follows the flow of the program until it reaches a

```
GOSUB line number
```

It notes the line in which this command occurs, then goes to the subroutine and executes the relevant operation. The subroutine ends with a

```
RETURN
```

which tells the computer to return to the main body of the program. Notice that RETURN is not followed by a line number: the computer uses the information it stored earlier to return to the line *following* that containing the GOSUB.



The GOSUB command on this computer has special features: its line number can be represented by a variable, or an expression. So you can have

```
GOSUB A
```

or

```
GOSUB 1000+A
```

and so on.

In addition GOSUB can be followed by a label – like this, for example:

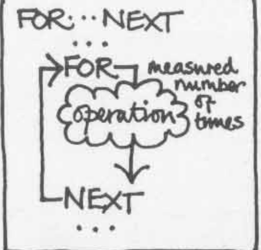
```
100 GOSUB LABEL name
```

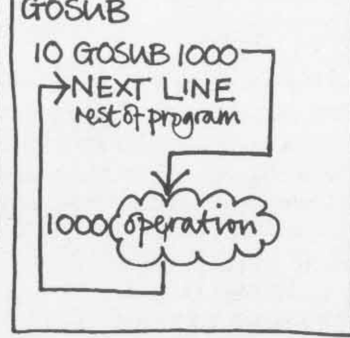
The name can be any length, within the limits of the maximum line length of 80 characters, but the shorter it is, the more efficiently the program will run because the computer will be able to find it faster.

The subroutine itself is labelled like this:

```
1000 LABEL name
1100 ...operation....
```

The label has two advantages: first, it is fast. Second, it allows you to write programs which are virtually independent of their line numbers.





The difference between GOTO, FOR...NEXT, and GOSUB.

Here's a program which demonstrates how subroutines work:

```
100 CLS [RETURN]
110 GOSUB LABEL G [RETURN]
120 PRINT "BACK FROM SUBROUTINE!" [RETURN]
130 PRINT "The key you pressed was ";G$ [RETURN]
140 GOTO 110 [RETURN]
150 REM [RETURN]
160 REM START OF SUBROUTINE [RETURN]
170 LABEL G
180 PRINT "EXECUTING SUBROUTINE..." [RETURN]
190 PAUSE 10000 [RETURN]
200 PRINT "Press a key!" [RETURN]
210 LET G$=GET$ [RETURN]
220 RETURN [RETURN]
230 REM END OF SUBROUTINE [RETURN]
RUN [RETURN]
```

`GOTO` can also be followed by a label.

A complex program can consist of a short 'organising' main program and a number of clearly labelled subroutines.

PROCEDURES

Procedures are similar to subroutines, in that they form distinct parts of the program and contain operations which can be called up by the main program. They can be used many times and by different parts of the main program.

They are controlled by three commands:

```
PROC name
DEFPROC name
ENDPROC
```

```
DEFPROC
```

marks the beginning of the procedure, and is followed by a name, which acts as a label, to distinguish one procedure from another. The name must not contain brackets – we will see why in moment. The end of the procedure is marked by

```
ENDPROC
```

The computer is told to execute the procedure by the command

```
PROC name
```

in the main part of the program. (This is the equivalent of a `GOSUB` command).

The computer recognises spaces as part of a procedure name, (eg `DEFPROC setting up the board`). If you accidentally add spaces to the end of a `PROC` call it will treat it as a different name, and will not be able to find the `DEFPROC`.

The procedure must be positioned so that the computer cannot run it except through a `PROC` call: that is, it must follow an `END`, or a `GOTO` which sends the computer back to the beginning of the program. If it finds a `DEFPROC` during a program run, the computer will display a `Wrong mode` error message.

Procedures are different from subroutines because you can pass **parameters**, values of variables, from the main program into the procedure. When you define the procedure you can follow the `DEFPROC` with the names of the variables you want to pass values to, like this:

```
DEFPROC name (A,B,C)
```

The variable names must be in brackets (which is why the procedure name must not contain brackets) and must be separated by commas. These variable names are called the **formal parameters**.

The values you want to assign to these variables (the **actual parameters**) are then included in the `PROC` command

```
PROC name (10, a*b, RND*100)
```

Again, the values must be placed in brackets and separated by commas.

If you include an undefined variable in the parameters you are passing, the error message will report that the error exists in the line containing the `DEFPROC` command, when in fact it is in the line containing the `PROC` call.

We have already used the `FOR...NEXT` loop, and seen how versatile it is. Lynx Basic also allows you to use two other types of loop.

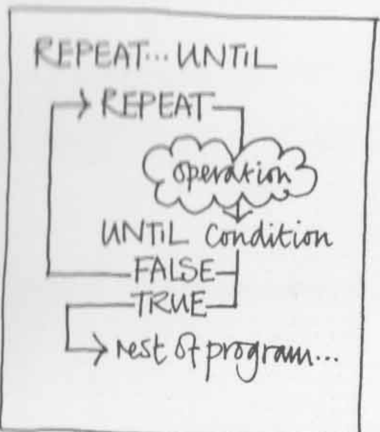
REPEAT...UNTIL

`REPEAT...UNTIL` has this format:

```
100 REPEAT
110 operation
120 UNTIL condition
```

The operation is repeated until the condition is fulfilled. When it is, the computer continues to execute the rest of the program.

Because the computer tests the condition at the *end* of the loop, the operation is always performed *at least once*.



Here is an example of `REPEAT...UNTIL` in action: a guessing game.

```
10 REM GUESS A NUMBER [RETURN]
20 R= INT(10*RND)+1 [RETURN]
30 C=0 [RETURN]
40 REPEAT [RETURN]
50    LET C=C+1 [RETURN]
60    INPUT "What is your guess";G [RETURN]
70    PRINT "You were"; ABS (G-R);"out!" [RETURN]
80 UNTIL G=R [RETURN]
90 PRINT "It took you";C;"goes!" [RETURN]
100 END [RETURN]
RUN [RETURN]
```

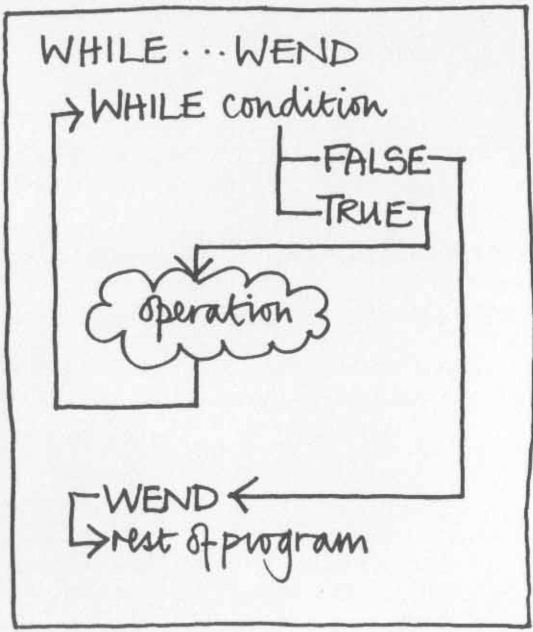
WHILE...WEND

`WHILE...WEND` also creates a loop. Its format is this:

```
100 WHILE condition
110 operation
120 WEND
```

If the condition is true, the loop is executed until it becomes false.

Notice that the condition is tested at the *beginning* of the loop. If it is false the computer skips to `WEND`, and executes whatever follows. This means that, unlike the `REPEAT...UNTIL` loop, a `WHILE...WEND` operation may not be performed at all.



Here is an example of `WHILE...WEND`: a program which prints numbers from 1 to 255 in decimal, hexadecimal and binary.

```
10 DIM A(7) [RETURN]
100 FOR J=0 TO 7 [RETURN]
110  LET A(J)=0 [RETURN]
120 NEXT J [RETURN]
125 FOR I=1 TO 255 [RETURN]
130  LET J=0 [RETURN]
140  LET A(0)=A(0)+1 [RETURN]
150  WHILE A(J)=2 [RETURN]
160       LET A(J)=0,J=J+1,A(J)=A(J)+1 [RETURN]
170  WEND [RETURN]
175  PRINT I," ",#I," ", [RETURN]
180  FOR J=7 TO 0 STEP -1 [RETURN]
190       PRINT A(J); [RETURN]
200  NEXT J [RETURN]
205  PRINT [RETURN]
210 NEXT I [RETURN]
RUN [RETURN]
```

`FOR...NEXT`, `REPEAT...UNTIL` and `WHILE...WEND` can all be combined and **nested**.

TRUE and FALSE

The Lynx has two functions which can be used to turn `REPEAT...UNTIL` or a `WHILE...WEND` into a continuous loop. These are

`TRUE` which gives value of 1

and

`FALSE` which gives a value of 0

They can be used like this:

```
REPEAT
operation
UNTIL FALSE
```

and are really intended to make the construction neat and legible.

ERROR

`ERROR` is a command which allows you to call up the Lynx's own error messages from within your program.

It has this format

```
ERROR number
```

(the code number of each error mesage is listed in Appendix 1).

Chapter 12: FURTHER MATHS

In addition to those we saw in Chapter 3, the Lynx has other mathematical capabilities.

When it prints out very large or very small numbers, the Lynx uses **scientific notation**.

For example, one million (1,000,000) in scientific notation is

```
1 E+6
```

that is, 1*10**6, or 1 followed by six 0s; and 0.0001 is

```
1 E-4
```

You can enter numbers into the computer in scientific notation.

ROUND and TRAIL

Throughout a calculation, the computer works to an accuracy of eight digits, but when it prints the final value on the screen it automatically rounds it, either up or down, to six digits. You can turn off this rounding by typing

```
ROUND OFF
```

and back on again by typing

```
ROUND ON
```

In certain circumstances, you may want to be sure of the accuracy of a number. In this case you can use TRAIL, which tells the computer to print the number adding trailing zeros to bring it up to an accuracy of eight digits if rounding is off, or six if it is on. To use TRAIL, type

```
TRAIL ON
```

to turn it off, type

```
TRAIL OFF
```

PARTS OF VALUES

Sometimes, you may want to use only part of a number or variable, say the integer (whole number) part. The Lynx has a range of functions which allow you to select parts of a value. These all have the normal function format

```
function name (X)
```

where X represents the number or variable you want to process (the argument of the function). It must be placed in brackets.

INT

INT is probably the most useful of these: it tells the computer that you want only the integer part of the number or variable. So

```
INT (21.3650)
```

is 21.

INT does not round the number up.

FRAC

FRAC gives you the fractional part of a number:

```
FRAC (5.3698)
```

is 0.3698.

ABS

ABS cuts the sign off a number or variable and gives you just the digits, the **absolute** value. So if the value of A is -10,

```
ABS (A)
```

is 10.

SGN

SGN gives you the sign of the number or variable, expressed as either 1 or -1.

SGN(10) will return 1

SGN(-10) will return -1

Zero is a special case:

SGN(0) returns 0.

INF

INF returns as its value 9.9999999 E+63, which is the closest number to infinity the Lynx is able to process.

If you call INF when rounding is on, it will be rounded up to a number higher than the computer can process, and it will display an error message.

ARCSIN, ARCCOS, ARCTAN

ARCSIN, ARCCOS, and ARCTAN take, respectively, the sine, the cosine, and the tangent of an angle as their argument, and return the value of the angle in radians. They have the usual function format:

```
ARCSIN (sine)
ARCCOS (cosine)
ARCTAN (tangent)
```

DIV

DIV is an integer division operator:

3 DIV 2 gives 1.

It has the same position in the algebraic hierarchy as division and multiplication.

MOD

`MOD` is a modular arithmetic operator:

`5 MOD 4` gives 1.

It has the same position in the algebraic hierarchy as division and multiplication.

FACTORIALS

`FACT (X)`

returns the factorial of X (the factorial of 4, for example, is 1*2*3*4). The function uses the integer part of X only.

EXPONENTIALS and NATURAL LOGS

`EXP (X)`

returns the value of **e** raised to the power of X.

`LN (X)`

returns the log to the base **e** of X.

Chapter 13: THE PRINTER

If you have a Lynx printer or a Lynx interface, you will be able to follow the connection instructions that come with it.

If you have your own printer and want to make your own interface, consult the technical information on the External Connections sheet (Appendix 4)

Lynx Basic has three printer commands.

LLIST

`LLIST` is very similar to `LIST`: it tells the computer to list your program, but to the printer.

It can be very useful to have a 'hard copy' of your program: it enables you to see the entire program at once, to trace the flow of the program, and to note down any changes which need to be made.

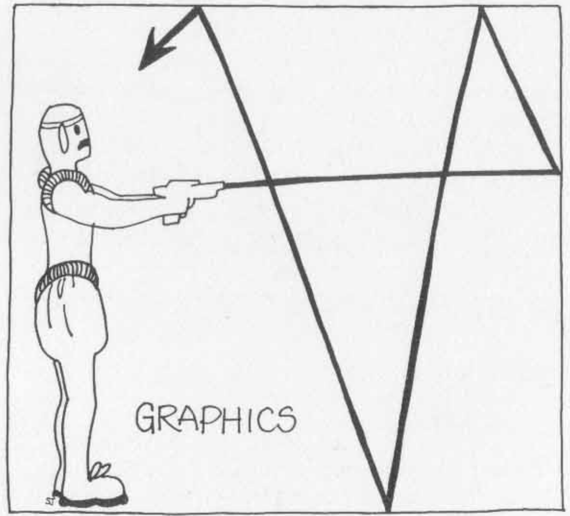
LPRINT

Similarly, `LPRINT` is the the printer equivalent of `PRINT`.

LINK

`LINK` is a special command which tells the computer to make, simultaneously, a printed copy of anything which is displayed on the screen.

Chapter 14: GRAPHICS AND BEEP



THE COLOURS

The Lynx can display eight different colours on the screen. Each colour has a number code, as follows:

```
0-BLACK
1-BLUE
2-RED
3-MAGENTA
4-GREEN
5-CYAN
6-YELLOW
7-WHITE
```

INK and PAPER

You can specify the background colour of the screen using either

```
PAPER colour number
```

or

```
PAPER colour name
```

You could type either PAPER 3 or PAPER MAGENTA, for example.

And you can specify the colour of the writing (and the graphics characters) using

```
INK colour number
```

or

```
INK colour name
```

All eight colours can be on the screen at the same time. Try this demonstration:

```
10 PRINT "*";[RETURN]
20 INK RAND (8) [RETURN]
30 PAPER RAND (8) [RETURN]
40 IF INK=PAPER THEN GOTO 30 [RETURN]
50 GOTO 10 [RETURN]
RUN [RETURN]
```


INK and PAPER are also functions: they return the code number of the current ink or paper colour.

PROTECT

There are three primary colours: RED, BLUE and GREEN. YELLOW is made by adding red and green together, CYAN by adding green and blue, and MAGENTA by adding blue and red. WHITE is made up from all the primaries, BLACK from none of them.

The Lynx has three banks of memory allocated to handling colours, one for red, one for blue, and one for green: the other colours are created by combinations. It is possible, however, to stop the Lynx using one, two, or all three of these banks of memory, and prevent it from using particular colours, using PROTECT.

```
PROTECT colour number/name
```

stops the Lynx using the bank of memory specified, so

```
PROTECT 1
```

will stop it using BLUE. Anything already on the screen coloured blue will then be 'protected', and cannot be overwritten or erased, even by using CLS. You will not be able to write in blue, or in colours which include blue: if you try to write in CYAN, for example, only the green component will appear.

If you protect a secondary colour – for example,

```
PROTECT MAGENTA
```

you will disable two primary banks, BLUE and RED.

If you protect WHITE you will disable all three, and nothing will be written on or erased from the screen.

You can enable all the banks again by protecting BLACK (disabling none of them).

PROTECT is a very useful command, for two reasons. First, it means that you can draw a background in one colour and protect it, then use other colours in the foreground, leaving the background intact.

Second, because the computer is handling less memory when one or two banks are disabled, it works much faster. So if you PROTECT MAGENTA, you can work in green at approximately twice normal speed.

THE GRAPHICS CHARACTERS

There are 32 graphics characters stored in the computer's memory, which can be printed on the screen using

```
PRINT CHR$ code number
```

or can be typed in through the keyboard if the computer is in graphics mode. To put it in graphics mode, first make sure that the [SHIFT LOCK] is in upper case, then type

```
[CONTROL] 1 [RETURN]
```

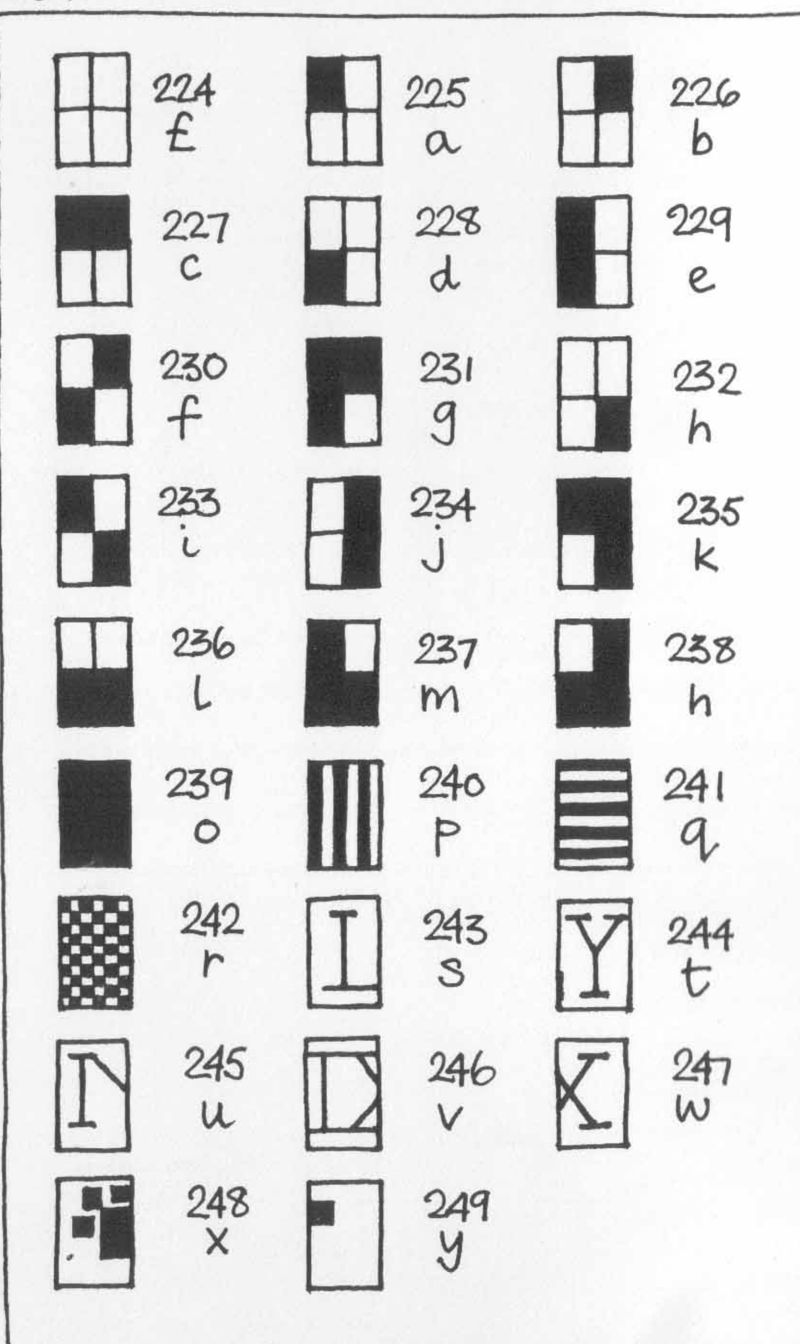
then press [SHIFT] and the appropriate key (see diagram).

To exit graphics mode, type

```
[CONTROL] 1 [RETURN]
```

again.

The graphics characters are



You can use numbers 224 to 242 to build up simple shapes and textures, and number 242 can be used to 'mix' colours:

```
100 REM CHECK BLANKET [RETURN]
110 CLS [RETURN]
120 FOR J=0 TO 7 [RETURN]
130 PAPER J [RETURN]
140 FOR I=0 TO 2 [RETURN]
150 FOR K=0 TO 7 [RETURN]
160 INK K [RETURN]
170 PRINT CHR$(242);CHR$(242);CHR$(242);CHR$(242);CHR$(242); [RETURN]
180 NEXT K [RETURN]
190 NEXT I [RETURN]
200 NEXT J [RETURN]
210 LET X=GETN [RETURN]
220 PAPER 0 [RETURN]
RUN [RETURN]
```

Characters 243 to 249, when printed side by side, form a Lynx logo.

HIGH RESOLUTION GRAPHICS

THE SCREEN

Screen size is measured in dots or **pixels**; the Lynx's screen measures 256 * 248 pixels. Every one of these dots can be processed: each one is 'labelled' by a co-ordinate which describes its position on the screen.

In the computer's memory, the screen is divided into a grid. Starting at the top left-hand corner, the columns of this grid are numbered from 0 to 255, and the rows from 0 to 247. You can refer to each individual dot by its co-ordinate:

```
column number, row number
```

The co-ordinates can be expressed as numbers, variables or expressions.

THE CURSOR

When you enter text into the computer, your position on the screen is marked by a cursor. The Lynx also has a graphics cursor but, unlike the text cursor, this is not a visible symbol; instead, it is a **position**, stored in the computer's memory.

All the graphics commands involve moving the cursor to a new position, which you specify using the screen co-ordinates.

MOVE

`MOVE` has this format:

```
MOVE co-ordinate, co-ordinate
```

It simply moves the cursor, *from* wherever it was, *to* the position specified by the co-ordinates (it does not draw a line).

DRAW

`DRAW` has this format:

```
DRAW co-ordinate, co-ordinate
```

It is similar to `MOVE`, except that it draws a line as it moves the cursor, drawing in the current `INK` colour.

Here is a program which uses MOVE to set up a cursor position, then draws a line to co-ordinates generated by the equations in lines 150 and 160, then uses MOVE again to shift to the next position (the result is very beautiful):

```
100 CLS [RETURN]
110 FOR J=0 TO 360 STEP 10 [RETURN]
120 MOVE 100, 60 [RETURN]
130 LET X=SIN(RAD(J)) [RETURN]
140 LET Y=COS(RAD(J)) [RETURN]
150 DRAW 100-60*Y,60+30*X [RETURN]
160 DRAW 100+60*X,60+30*Y [RETURN]
170 DRAW 100,200 [RETURN]
180 NEXT J [RETURN]
RUN [RETURN]
```

Try changing line 160 to

```
160 DRAW 100+60*X,200+30*Y [RETURN]
```

DOT

DOT co-ordinate, co-ordinate

draws a dot in the current ink colour at the specified co-ordinates. You can colour every position on the screen individually: here is a program which draws dots in random colours at random positions on the screen:

```
100 REM RANDOM DOTS [RETURN]
110 PAPER BLACK [RETURN]
120 CLS [RETURN]
130 INK RAND(7)+1 [RETURN]
140 DOT RAND(240), RAND(240) [RETURN]
150 GOTO 130 [RETURN]
RUN [RETURN]
```

The next program draws dots at co-ordinates generated by a parametric equation:

```
100 REM RANDOM SPIRAL [RETURN]
110 PAPER BLACK [RETURN]
120 CLS [RETURN]
130 LET R=RAND(1440) [RETURN]
140 INK RAND(6)+1 [RETURN]
150 DOT 100+R/20*COS(RAD(R)),100+R/24*SIN(RAD(R)) [RETURN]
160 GOTO 120 [RETURN]
RUN [RETURN]
```

PLOT

PLOT combines all the features of the three graphics commands we have just explored. It has this format:

```
PLOT mode number, co-ordinate, co-ordinate
```

It has five different modes:

0 is the same as a MOVE

1 is a relative move: the co-ordinates represent the amount by which the cursor is moved, not its final position.

2 is the same as DRAW

3 is a relative draw: again, the co-ordinates represent the amount by which the cursor is moved.

4 is the same as DOT

WINDOW and PRINT@

These two commands are used for handling character strings, so the co-ordinates they use are determined by the size of a character block (six pixels across x 10 pixels down). If you choose a different multiple of pixels, characters will fall off one side of the screen and reappear on the other side (**wraparound**).

Using WINDOW, you can tell the computer to print on a particular area of the screen only: anything displayed on the portion of the screen outside the window is left uncorrupted.

You use screen co-ordinates to specify the position of the window,

0-126 columns

0-248 rows

(Note that with WINDOW and PRINT@, the columns are 2 pixels wide).

If you are using the window to display text, your column co-ordinate should be a multiple of three (six pixels) and the row a multiple of 10.

WINDOW has this format:

```
WINDOW first column,last column+1,
        first row,last row+1
```

The Lynx's normal text window is

```
3,123,5,245
```

so to revert to using the full screen area, use

```
WINDOW 3,123,5,245
```

If the cursor is not inside the window when you set it up, you can move it there by typing

```
PRINT CHR$ 23
```

or

```
VDU 23
```

(see CONTROL CODES, later in this chapter).

Alternatively, use the arrow keys: the cursor will jump into the window when it reaches one of the bounds. Or you can use CLS, which will clear the entire screen, inside and outside the window, but will home the cursor to the top left-hand corner of the window.

You can move the cursor out of the window using PRINT@.

PRINT@

You can use

```
PRINT@ column number, row number;.....
```

(roughly 0-124 columns, 0-240 rows) to position character strings on the screen. Again, the co-ordinates you choose will be affected by the size of character block.

Here's a simple but attractive demonstration of PRINT@:

```
100 CLS [RETURN]
110 PAPER BLACK [RETURN]
120 INK RAND(7)+1 [RETURN]
130 PRINT@ RAND(125), RAND(240);"*"; [RETURN]
140 GOTO 120 [RETURN]
RUN [RETURN]
```

POS and VPOS

POS and VPOS are functions which return the position of the cursor; POS gives the column number, and VPOS the row number.

Like WINDOW and PRINT@, POS uses columns which are two pixels wide.

BEEP

BEEP tells the computer to make a beeping sound. It has this format:

```
BEEP wavelength, number of cycles, volume
```

The wavelength can be between 0 and 65535 – the shorter the wave length the higher the beep.

The number of cycles sets the length of the beep. It can be between 0 and 65535. The higher the wavelength is, the shorter the cycle is, so to get a high beep and low beep of the same length you would need to make the number of cycles in the first much higher than in the second – that is,

```
wavelength*number of cycles
```

should be the same.

The volume can be set between 0 and 63.

Experiment with different values; try these to start with:

```
BEEP 50,1000,63
BEEP 200,1000,63
BEEP 5000,10,53
```

You can always stop the computer in mid-beep using *[ESC]*.

(See also SOUND, Chapter 16).

CHANGING THE CURSOR

CCHAR

You can alter the text cursor.

The cursor is made up of two characters, printed alternately. The normal Lynx cursor alternates between a block and a space, but you can specify different characters, using two characters, a character and a space, or – if you want to stop the flashing – the same character twice.

To redefine the cursor, first choose the characters you want and use their ASCII codes like this:

```
CCHAR first code*256+second code
```

Alternatively, you can convert the codes to hexadecimal (see Chapter 16) and type:

```
CCHAR &hex code hex code
```

with no space between the two codes. So, if you wanted the cursor to alternate between a space and #, for example, you would take their codes, 32 and 35 respectively, convert them to hex, 20 and 23 and enter

```
CCHAR &2320
```

CFR

You can alter the rate at which the cursor flashes, using

```
CFR number between 0 and 65535
```


1 is fastest
65535 is very slow
0 is slowest

The normal flashing rate is about 500.

CONTROL CODES: PRINT CHR$ and VDU

In Chapter 7 we examined CHR$ and saw that it was used to convert the ASCII code of a character into the character string itself. There are 256 possible codes (because computer memory works in bytes, and the highest binary number that can be stored in a group of eight storage spaces is a series of eight 1s, 11111111, which in decimal is 255).

The American Standard Code for Information Interchange (ASCII) allocates numbers 32 to 127 to the normal computer character set, which includes all the letters of the alphabet, upper and lower case, ! @ # $ % and so on, and the Lynx follows this standard. In addition, on the Lynx, codes 128 to 223 represent another copy of the character set, which can be modified into **user defined graphics** (see later in this chapter) and codes 224 to 249 represent the **graphics characters** (described earlier in this chapter).

Codes 0 to 31 are used to represent not characters, but cursor movements and other graphics and sound commands; or sometimes combinations of them. You can use them either like this:

```
PRINT CHR$ (code number)
```

or like this:

```
VDU code number
```

CHR$ is a string function, and can be used to build up a string, like this:

```
100 CLS [RETURN]
110 LET A$=CHR$(24)+"LYNX" [RETURN]
120 FOR J=1 TO 9 [RETURN]
130 PRINT A$ [RETURN]
140 NEXT J [RETURN]
150 PAUSE 10000 [RETURN]
160 PRINT CHR$(25) [RETURN]
RUN [RETURN]
```

Code 24 tells the computer to print double size characters, code 25 restores it to normal.

VDU is a command. It can be used to build up a whole series of screen and cursor commands:

```
100 VDU 4,18,24 [RETURN]
110 INPUT "What is your name";A$ [RETURN]
120 PRINT "HELLO ";A$; [RETURN]
130 VDU 18,25,10,10,10 [RETURN]
RUN [RETURN]
```

Code 21 (overwrite) allows some very interesting effects:

```
100 VDU 4,21 [RETURN]
110 REPEAT [RETURN]
120 VDU 1,RAND(6)+1, RAND(96)+32 [RETURN]
130 UNTIL KEY$="S"[RETURN]
140 VDU 20 [RETURN]
RUN [RETURN]
```

Let the program run for a few screens; you can stop it by pressing S.

All the codes are listed below:

0 not implemented

1 eg `VDU 1, colour number`
changes `INK` to specified colour

2 eg `VDU 2, colour number`
changes `PAPER` to specified colour

3 not implemented

4 clears screen and homes cursor

5 moves cursor up one pixel line

6 moves cursor down one pixel line

7 beeps

8 backspace and erase character

9 tabs cursor to next field

10 line feed (moves cursor down 10 pixel lines)

11 not implemented

12 moves cursor one character block to the right

13 carriage return, line feed, clear to end of line

14 turns cursor on

15 turns cursor off

16 moves cursor to top of screen

17 not implemented

18 swaps `INK` and `PAPER` colours (**inverse video**)

19 carriage return and line feed if cursor is not at beginning of line

20 overwrite off

21 overwrite on

22 backspaces cursor

23 homes cursor (takes cursor inside window)

24 turns double height characters on

25 turns double height characters off

26 not implemented

27 not implemented

28 moves cursor up 3 pixel lines (superscript)

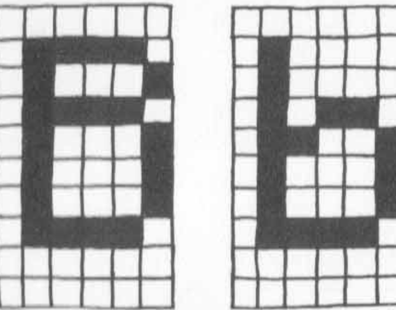
29 moves cursor down 3 pixel lines (subscript)

30 clears to end of line

31 carriage return, line feed

USER-DEFINED CHARACTERS

Character blocks on the Lynx measure six pixels across and 10 pixels down, and are made up of a pattern of dots and spaces, like this:



Each horizontal line of a character takes up a byte of storage, so an entire character takes up 10 bytes.

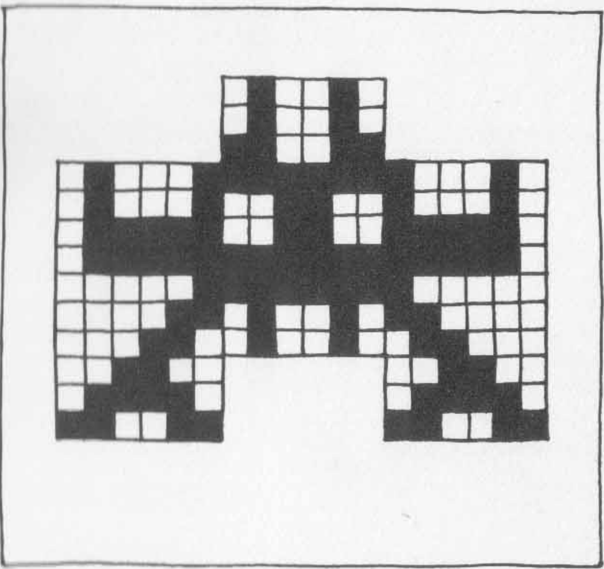
If you take a piece of squared paper, mark out a rectangle of 6x10 squares, then draw a pattern by filling in squares inside the rectangle, you will have designed your own character, which you can then feed into the computer.

Remember that if you are designing something which needs a space between itself and the next character, like a letter, you will need to incorporate the space into the design. On the Lynx, the first column, the top row, and the bottom 2 rows of a character are generally left blank (these make the spaces between the letters and the lines when text is displayed).

But if you are designing a graphics block you may want it to fill the whole space.

Once you have your design you can program it into the computer. There are several functions to help you do this: `ALPHA`, `GRAPHIC`, `LETTER` and `BIN`.

To see how they work, let's take an example. Let's suppose we want to define an 'invader'. First we need to draw out the design on squared paper:



Notice that the invader is made up of three characters, and that the middle character needs to be printed three pixel lines higher that the other two.

Here's how to program the invader into the computer:

```
100 RESERVE HIMEM-30
110 DPOKE GRAPHIC, HIMEM
120 FOR J=0 TO 29
130 READ A
140 POKE LETTER(128)+J, BIN(A)
150 NEXT J
160 DATA 010001,010001,011111
170 DATA 011111,000001,000011
180 DATA 000110,001100,011110
190 DATA 110011
200 DATA 010010,010010,011110
210 DATA 111111,001100,001100
220 DATA 111111,111111,010010
230 DATA 010010
240 DATA 100010,100010,111110
250 DATA 111110,100000,110000
260 DATA 011000,001100,011110
270 DATA 110011
280 CLS
290 FOR J=1 TO 11
300 READ A
310 LET A$=A$+CHR$(A)
320 NEXT J
330 DATA 24,1,2,128,28,129,29,130,25,1,7
400 PRINT@ 60,60;A$;
```

First we need to set aside memory for storing the characters. This is done using `RESERVE` and `HIMEM`, in line 100 (for more about these, see Chapter 16). A character takes up 10 bytes of memory, so we need to reserve 30 bytes.

The Lynx has two 'pointers' stored in its memory, which tell it where its character set is stored: `ALPHA` and `GRAPHIC`. These are two-byte locations which store the address of the beginning of the standard character set and the address of the duplicate character set (128 to 224) repectively. `ALPHA` could be used to alter the normal character set to incorporate accents, for example. But `GRAPHIC` is the pointer used for defining extra characters.

Line 110 makes `GRAPHIC` point to the extra memory we have set aside.

The design of the characters is stored as a series of 10 lines of six `0`s and `1`s, (`1` represents a shaded square, `0` a blank square). `BIN`, used in line 140, is a function which persuades the computer to treat the `0`s and `1`s as a binary number.

These numbers are 'poked' (see Chapter 16) into the reserved memory by line 140. `LETTER` is a function which uses the value of `GRAPHIC` to calculate the address of the first line of the character specified – in this case, number 128. The `FOR...NEXT` loop adds one to this each time it runs, so the values are poked into 30 consecutive bytes.

Now the characters have been defined, lines 280 to 400 tell the computer how to use them. Line 310 builds up a character string from the data stored in line 330, using some of the control codes we explored earlier. In particular, note the use of codes 28 and 29 to print the middle character higher than the other two.

You might like to change the end of the program to

```
400 FOR J=10 to 90 [RETURN]
410 PRINT@ 60,J;A$; [RETURN]
420 NEXT J [RETURN]
```

and make the invader move.

Then try changing line 410 to

```
410 PRINT@ J,J;A$;
```

or

```
410 PRINT@ 40,J;A$;@ 60,J;A$;@ 80,J;A$;
```

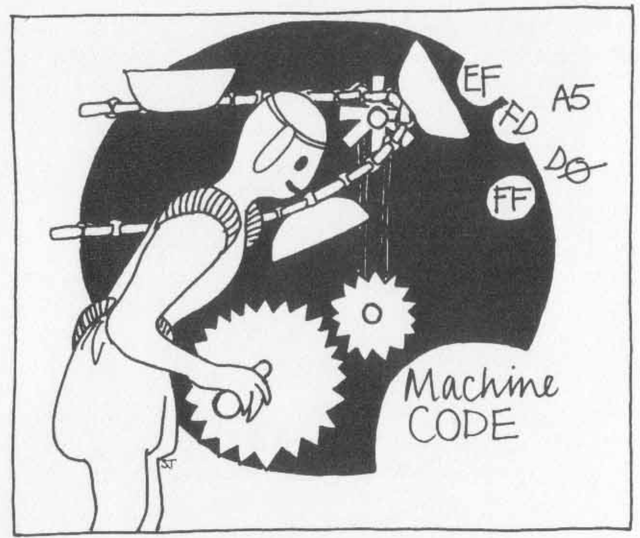
Every time the program runs it reserves more memory: eventually the computer will run out of memory. You can get round this by running the program from line 110.

IDEAS and EXAMPLES

Watch this!

```
100 LET B$="*"+CHR$(22)+CHR$(5) [RETURN]
110 PRINT@20,240; [RETURN]
120 FOR J=0 TO 220 [RETURN]
130  PRINT B$ [RETURN]
140 NEXT J [RETURN]
RUN [RETURN]
```

Chapter 15: WHAT IS MACHINE CODE?



This chapter is intended for people who know nothing about machine code. If you are already familiar with it, you can skip to the next chapter, which describes the Lynx's resident monitor, and several interesting Basic commands.

Machine code programming is a complex subject; the Lynx computer uses a Z80 microprocessor and so you will need to refer to a Z80 programming manual. But this chapter is intended to give you some idea of what machine code is.

Machine code has some important advantages: first, it is very flexible: you can use it to do things not possible in Basic; second, and perhaps most important of all, it runs very fast – and that can be very useful, especially in programs using graphics.

The microprocessor performs operations using a very simple language, called machine code. In the introduction we saw that the microprocessor could do nothing on its own, but was provided with instructions by a resident program which is written in machine code and stored in ROM. When your computer is on, it constantly runs this program, which is called an **interpreter**. It is the interpreter which allows you to write your programs in a language other than machine code: when you run your program, the computer is in fact running its interpreter and using your program as data.

On the Lynx, the interpreter allows you to write in Basic; though it is possible to obtain interpreters which will allow you to use other languages.

The Lynx's interpreter also allows you to include small amounts of machine code in your programs – see the special commands `CODE`, `LCTN`, and `CALL` described in the next chapter.

In addition, the Lynx has a **machine code monitor**, which provides you with facilities for writing, running, saving, and editing machine code similar to those available in Basic.

So what is machine code like?

The Z80 microprocessor has various specialised parts: it contains, for example, an Arithmetic-Logic-Unit which performs calculations, and an overall control unit which supervises the parts of the processor, sending the correct data to the correct place and so on.

But the most important features of the Z80 – to the programmer – are the **user registers**. A register is a single byte location within the Z80 itself, a place where data can be stored. The user registers are places which can be accessed by the programmer, and the majority of machine code programming consists of manipulating data in these registers.

The registers are generally used as register pairs. Their names are

```
AF  HL  DE  BC
AF' HL' DE' BC' IX IY SP PC
```

Some of the register pairs have specialised functions and are used for manipulating a particular type of data.

Try calling up the Lynx's machine code monitor – type:

```
MON [RETURN]
```

and the computer will display the value stored in each of these register pairs.

If you type `S [RETURN]` the computer will clear the screen.

Now if you type `H [RETURN]`, the computer will display a table of the contents of part of its memory. The numbers in the middle part of the table, like `C9`, `4C` and so on, are machine code instructions.

You can **quit the monitor** by typing `J [RETURN]`.

The Z80 recognises a finite number of instructions:about 500. Each instruction is very specific and is represented by a code number. The processor recognises the code and executes the appropriate operation.

For convenience, the programmer enters these codes in **hex**: hexadecimal (base16) numbers.

DECIMAL	HEXADECIMAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
100	64
1000	3E8
10000	2710

At first sight hexadecimal may seem like an odd choice, but it is very convenient. A single byte can be represented by two hex digits: the right-hand digit represents the right-hand four bits, the left-hand digit represents the left-hand four.
Each of the 500 or so Z80 commands, plus any data they require, can be represented by a hex number or numbers. It is in this form that machine code must be typed – either into a `CODE` line in a Basic program, or into memory using the machine code monitor.

If you want to know more about the individual commands and their codes, you will need to consult a Z80 programming manual.

ASSEMBLY LANGUAGE

Writing machine code in hexadecimal numbers is a laborious task. But using a program – similar to the Basic interpreter – called an **assembler**, it is possible to program in machine code using **mnemonics**, which look like this:

ADD HL,DE

(this adds the contents of registers HL and DE together and stores the result in HL).

Chapter 16 (10H): MACHINE CODE

This chapter is intended for people who already know something about machine code and how to use it. It explores the Basic commands available on the Lynx for incorporating machine code into Basic programs, and also lists the commands available on the Lynx's machine code monitor.

HEX NUMBERS: PRINT # and &

Using

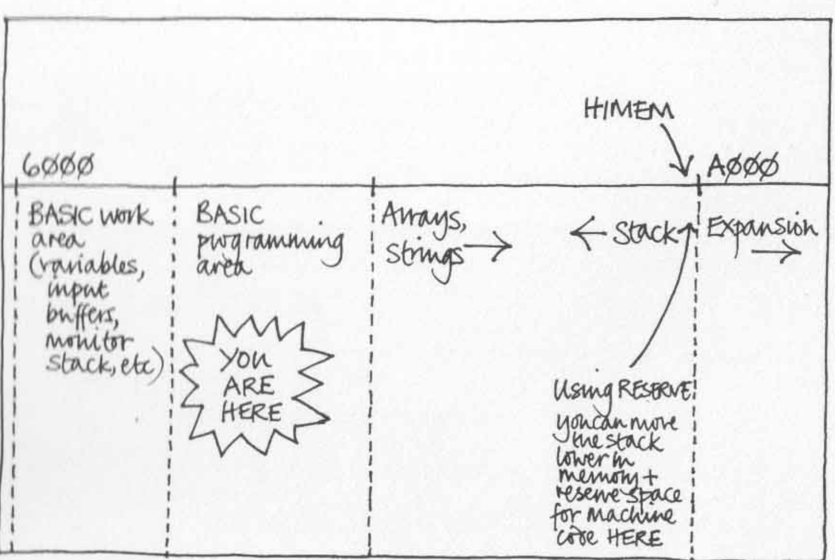
```
PRINT # expression
```

you can tell the computer to print the expression in hex. It will display an `H` after the number, like this:

```
A02FH
```

You can input a hex number if you mark it with an `&`, like this:

```
&A02F
```



PEEK and DPEEK

You can examine the contents of a memory location using `PEEK`.

```
PEEK (address)
```

the address must be in brackets.

```
DPEEK (address)
```

examines two adjacent locations, at the address and the address+1. It returns a single value,

```
256*(contents of address+1)+ contents of address
```

This is the equivalent of a

```
LD HL,(address)
```

POKE and DPOKE

You can insert information into a location using `POKE`. It has this format:

```
POKE address,value
```

The value can be between 0 and 9.9999999E7, but the computer will convert the value to modulo 256. The address can be between 0 and 9.9999999E7, but will be converted to modulo 65536 (64K).

```
DPOKE address, value
```

loads two adjacent locations with a value. The Least Significant Byte (LSB) is loaded into the address, and the Most Significant Byte (MSB) is loaded into the address+1.

This is equivalent to a

```
LD (address), HL
```

`DPOKE` is useful for changing **vectors**.

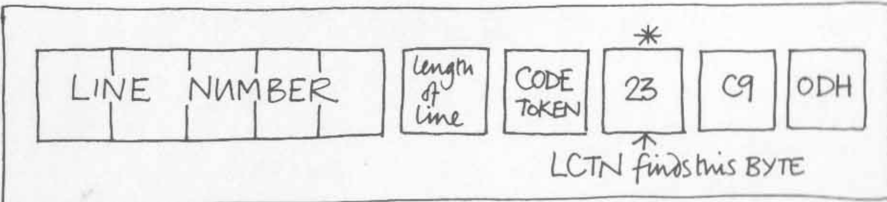
CODE, LCTN CALL and HL

These four commands make it easy for you to incorporate machine code routines in a Basic program.

`CODE` allows you to store machine code in an ordinary program line; and allows you to type it in directly. A line will look like this:

```
10 CODE 23 C9
```

The code is typed in as a series of hex values, separated by spaces, *but not, in this case, marked by an &.* As the line is stored in memory, the computer converts the values into their binary equivalents and stores them in successive bytes. In memory the line above would look like this:



The line number takes up 5 bytes, and the line length is stored in the following byte. Next comes the command, `CODE`, stored as a token. The following bytes store the hex numbers. `ODH` is a carriage return, which marks the end of the line.

`CODE` allows you to **store** the machine code: ordinarily the computer will ignore a `CODE` line, (like a `REM` statement). There are other Basic commands which enable you to tell the computer to execute the stored code.

```
LCTN (line number)
```

tells you the address of the first byte following the command token of the line you specify (this is marked by a star in the example above). Using it, you can poke values into program lines. For example

```
POKE LCTN (10),7
```

would change the contents of the starred byte (in the diagram above) from 23 to 7.

The most important use of `LCTN`, however, is with `CODE` and `CALL`.

```
CALL address
```

`CALL` is the machine code equivalent of `GOSUB`: it tells the computer to find and execute the machine code subroutine at the address specified. If you have stored the subroutine in `CODE` lines,

```
CALL LCTN (line number of CODE line)
```

sends the computer to the address in which the first byte of your subroutine is stored and tells it to execute the code.

If you want to use some part of the Basic program, the value of variable `V` for example, in the machine code subroutine, you can transfer it to the HL register using

```
CALL address, value
```

For example:

```
CALL address, V
```

would put the value of `V` into the HL register before the subroutine is executed.

HL

When the computer finishes executing the subroutine, it stores the final value of the HL register pair as a read only variable,`HL`. This new value can be returned to the Basic program.

HIMEM and RESERVE

`HIMEM` is a read-only variable which tells you the first free address after the stack (see the memory map): it tells you the lowest position in memory from which you can start to store machine code.

```
RESERVEexpression
```

allows you to move the stack to a lower memory location, to increase your machine code storage area. The expression is an address; in effect a new `HIMEM` value.

If you try to move the stack too far (so that it would corrupt other stored material) or if you try to move it to a higher location, you will be given an error message.

BINARY OPERATORS

The Lynx has three binary operators:

`BNAND` – binary AND

`BNOR` – binary OR

`BNXOR` – binary EXCLUSIVE OR

which allow 16-bit binary logical operations, like this:

```
A BNOR B
```

`BNAND` has the same priority as `AND`, `BNOR` and `BNXOR` have the same priority as `OR`.

INP and OUT

The microprocessor communicates with other parts of the computer, and with peripherals, through I/O (Input/Output) ports. Like the RAM, the values of these can be examined and altered.

```
INP(port)
```

allows you to examine the value of a Z80 port. It passes the argument to BC, then performs an

```
IN A,(C)
```

Note that A8-A15 contain the value of B during an `IN A,(C)`.

`OUT port,value`

sends a value to a specific port. It performs an

```
OUT (C),A
```

where `A` contains the specified value and BC the specified port. Note that A8-A15 contain the value of B during an `OUT (C),A`.

SOUND

`SOUND address,delay between outputs`

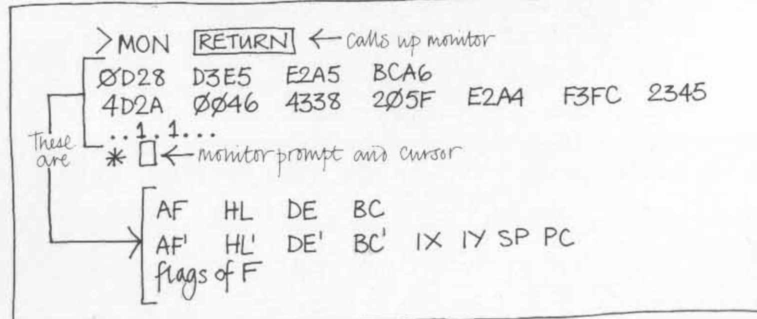
sends the computer to the address specified, and tells it to convert the values found from there onwards into sound. It will stop when it finds a value of 0.

The delay between outputs can be any number between 0 and 65535.

Using `RESERVE` and `HIMEM`, provided you have enough storage memory space and can work out the necesary values, you can build up sound effects (even synthesise speech!) by poking the values into memory. You can then run them, using `SOUND`.

THE MONITOR

The monitor can be called up by typing `MON [RETURN]`. Initially, it will display the contents of the Z80 registers, like this:



The contents of the registers are stored, then replaced when you leave the monitor. The special monitor prompt is a `*`.

If you make a mistake, using the wrong format for a particular command, the computer will display `????`.

Here are the monitor commands, in alphabetical order; the symbols X Y and Z represent hex numbers.

A: arithmetic

`A X Y`

`A` displays `X+Y X-Y ZZ`

where `ZZ` is the jump relative required to get from `X` to `Y`. If a jump is not possible the computer will display `??`.

Examples:

`A 145 111` will give `0256 0034 CA`

`A 2345 6789` will give `8ACE BBBC ??`

B: breakpoint

`B X B`

can be used to set a **breakpoint**. A breakpoint is a debugging tool which lets you break into a program. You set the breakpoint in a particular part of the program. When the computer reaches it, it stores the contents of the registers and passes control to the monitor. You can then examine the contents of the registers.

`B X` will set a breakpoint at `X`

The value `X` originally contained will be stored and can be restored to `X`, using

`B`

You cannot set two breakpoints: only the last one entered can be restored.

Examples:

`B A000` will set a breakpoint at `A000`.

`B` will restore it.

C: copy (see also I)

`C X Y Z`

`C` copies contents of memory from `X` to `Y` for `Z` bytes. It will copy the contents of address `X` into address `Y`, of address `X+1` into address `Y+1`, and so on.

If the two blocks overlap, the contents of block `X` will be corrupted.

Examples:

`C A000 A0001 100`

will copy the contents of `A000` to `A001`, `A001` to `A002`...so the contents of `A000` would be reproduced throughout `A000-A101`.

D: dump to cassette

`D X Y Z "name"`

`D` dumps to cassette the contents of the memory from `X` through to `Y` with a transfer address of `Z` and a specified name which must be in inverted commas.

The transfer address is the point in the program from which it will start to run (automatically) when it is loaded into the computer. If you do not want the program to run automatically, use a transfer address of 0.

The name can be any length.

Example:

`D A000 AFFF A000 "TEST"`

will dump memory from `A000` up to `AFFF` inclusive, with a transfer address of A000 and the name `TEST`.

E: execute

`E X E`

`E X` downloads the registers which were stored when you entered the monitor, then executes the code from address `X`.

`E` executes from the stored program counter. It can be used after a breakpoint.

Example:

`E A000` will execute from `A000`.

F: fill

F X Y Z

F will fill memory from X to Y inclusive with byte Z.

Example:

F A000 AFFF 7E

will fill from A000 to AFFF with 7E.

G: go

G X

G will execute code from X, like a subroutine call. You can use it to add commands to the monitor.

DE will point to the first byte after X in the G command, so you can pass parameters to the routine.

Examples:

G A000 will call subroutine at A000.

If at A000 was

```
LD A,"#" ; Load with #
JP DSPLY ; Jump to display subroutine
```

the G A000 will display a #.

If at A000 was

```
INC DE   ; inc to next byte
LD A,(DE);  get byte
JP DSPLY ;  display
```

the G A000 * will display a *G A000 7 will display a 7

(These examples are purely illustrative!).

H: hex dump

H X H

HX dumps memory from X onwards to screen as hex and ASCII. In the ASCII, bit 7 is reset.

H dumps from the last H or M, L, W etc.

Example:

H B1E0 may give

```
B1E0 D0 49 C9 4E 4B 45 59 D2        PIINKEYR
B1E8 4E 44 C8 4C D0 4F 53 D6        NDHLPOSV
```

for 16 lines...........

Non-displayable ASCII characters are represented by ....

I: intelligent copy

I X Y Z

I moves blocks of 2 bytes from X to Y intact. If the blocks overlap, the block starting at X will be moved without corruption to start at Y.

J: jump to Basic

J returns the computer to Basic.

L: locate

L X Y

L locates occurrences of byte Y, starting at X, through to the end of memory. The address of each occurrence will be displayed. It can be aborted by pressing *[ESC]*.

M: modify

M X Y1 Y2 Y3 Y4...YN

M X

M

M lets you examine the contents of ROM or RAM, and change the contents of RAM.

To use it, type

M, followed by the address you want to modify. The display will be in this format:

address contents cursor

If you want to change the contents, type in the appropriate hex value, and press *[RETURN]*. The computer will display the next byte.

You can type a series of hex values separated by spaces, and these will be stored in successive addresses.

If you do not want to change the contents, press *[RETURN]*, and the computer will display the next byte.

You can backspace by typing / *[RETURN]*.

You can quit modify by typing . *[RETURN]*.

Example:

(The slanted type on the left shows the computer's response).

```
*M A000 [RETURN]
A000 <F3> 2E [RETURN]
A001 <21> 3E [RETURN]
A002 <07> 4D [RETURN]
A003 <A0> 56 56 67 67 [RETURN]
A007 <E3> / [RETURN]
A006 <67> / [RETURN]
A005 <67> / [RETURN]
A004 <56> / [RETURN]
A003 <56> [RETURN]
A004 <56> [RETURN]
A005 <67> . [RETURN]

*M [RETURN]
A005 <67> 11 22 33 44 [RETURN]
A009 <22> . [RETURN]

*M A000 22 33 44 55 66 [RETURN]
A005 <11> . [RETURN]
```

O: output to port

O X Y

O outputs byte Y to port X (it is the equivalent of OUT X,Y in Basic), doing an

```
OUT (C),A
```

where BC=X, A=Y.

P: increment program counter

P X

P increments the stored program counter by X.

Examples:

P 1 increments the program counter by 1.

P FFFF decreases it by 1.

Q: query port

Q X

Q will display the input from port X: it does an

```
IN A,(C)
```

where BC=X

R: read tape

R "NAME"

R will read a file with given name from cassette. The name must be in inverted commas.

Examples

R "FRED"

will load file FRED.

S: screen clear

S clears the screen.

T: type into memory

T X text

T allows you to type ASCII into memory, to be stored from X.

Example:

T A000 LYNX

will store 4C at A000 4C=L
59 at A001 59=Y
4E at A003 4E=N
58 at A003 58=X

U: update register

U reg X

U allows you to change the value of the stored register pair specified to X.

Examples:

U AF 4027 will change stored AF to 4027

U DE' 2345 will change stored DE' to 2345

Valid registers are

```
AF HL DE BC AF' HL' DE' BC'
IX IY SP PC
```

V: verify

V X Y Z

V will verify that the block of Z bytes starting at X is the same as that starting at Y. The computer will display addresses of any discrepancies.

Example:

V A000 B000 200

will check that A000 to A1FF is the same as B000 to B1FF.

W: word (see also L)

W X Y

W will search for word Y from X to the end of memory. The computer will display the address of each occurrence.

Note that the computer will search for the LSB of Y at the address, and the MSB at the address+1.

X

X displays the contents of the stored registers and the flags of F, in this format:

```
AF   HL   DE   BC
AF'  HL'  DE'  BC'  IX   IY   SP   PC
```

Z

Z will display the stored registers and what they point to.

Appendix 1: ERROR MESSAGES

If you make a mistake whilst programming, the computer will tell you by displaying an **error message** and the number of the line in which the mistake occurred. The error messages are designed to be self-explanatory, but have been listed here in alphabetical order, each with its code number, and some with a short explanation.

`Bad tape 29` will appear if, when you ask it to verify a recording of a program, the computer finds that the recording is corrupt.

`Cannot continue 17` indicates that, since stopping the program with *[ESC]*, you have altered the program in some way, so that `CONT` cannot be used: you must restart the program using `RUN`.

`Divide by zero error 5` tells you that you have tried to divide by 0.

`ENDPROC without PROC 28`

`Function argument error 9` appears when the argument given to a function is outside the allowed bounds of the function: for example,

```
SQR(-1)
```

would produce an error message, because only positive numbers can form the argument of `SQR`.

`GOSUB without RETURN 27`

`Line, label or PROC not found 12` will appear if, for example, you have used a `GOTO` line number which does not exist.

`Line too long 16` appears as you enter a line and tells you that you have exceeded the maximum number of 240 characters in that line.

`Missing bracket 4` appears as you enter a line if it contains an unmatched bracket.

`NEXT without FOR 20`

`Number out of range 13`

`Out of data 18` tells you that the computer has read all the data available, but is being asked to read more.

`Out of memory 1` indicates that you have filled all the computer's RAM. It will occur if you are trying to load a program from cassette which is too long to fit into RAM.

`Overflow error 6` tells you that a number has occurred which is too large for the computer to process, perhaps generated within your program.

`Redimensioned array 11` warns you that you are trying to redimension an array.

`REPEAT without UNTIL 26`

`Return stack full 25` tells you you have nested `FOR...NEXT` loops, etc, too many times.

`RETURN without GOSUB 19`

`Something missing 8` appears if you have forgotten to type in an operator or an operand.

`String error 3` appears if, for example, you have a string longer than 127 characters.

`Subscript out of range 14` tells you that you are trying to use an array which is outside the range you set up in your array: for example, trying to use `A(12)` after

```
DIM A(10)
```

`Syntax error 7` appears when the structure of the line is not intelligible: that is, when it does not conform to the pattern the computer expects.

`Type mismatch 15` appears if the computer is expecting one thing and is given something else.

`Undefined variable 21` indicates that you have tried to process a variable which has not previously been assigned a value.

`UNTIL without REPEAT 22`

`WEND without WHILE 23`

`WHILE without WEND 24`

`Wrong mode 2` will appear if you are trying to do something in calculator mode which can only be done in program mode, and so on.

Appendix 2: SHORTHAND

The Lynx has a shorthand facility, to make typing in programs quick and easy.

Insted of typing in the entire command, you can type in an abbreviation, followed by a full stop. The shortened version must be long enough to distinguish the command from any similar command. For example,

`L.` cannot be used for `LIST`, because of `LET`, but `LI.` is sufficient.

`AUTO` can be shortened to `A.`

`FOR` can be shortened to `F.`

`REPEAT` needs `REP.`

and so on.

SINGLE KEY ENTRIES

In addition, some commands are represented by a single letter, and are entered in by holding down the [ESC] key and typing in the appropriate letter.

Wherever possible, a letter has been chosen because it is an abbreviation of the command.

You may find that the most convenient policy is to learn and use just a few of them.

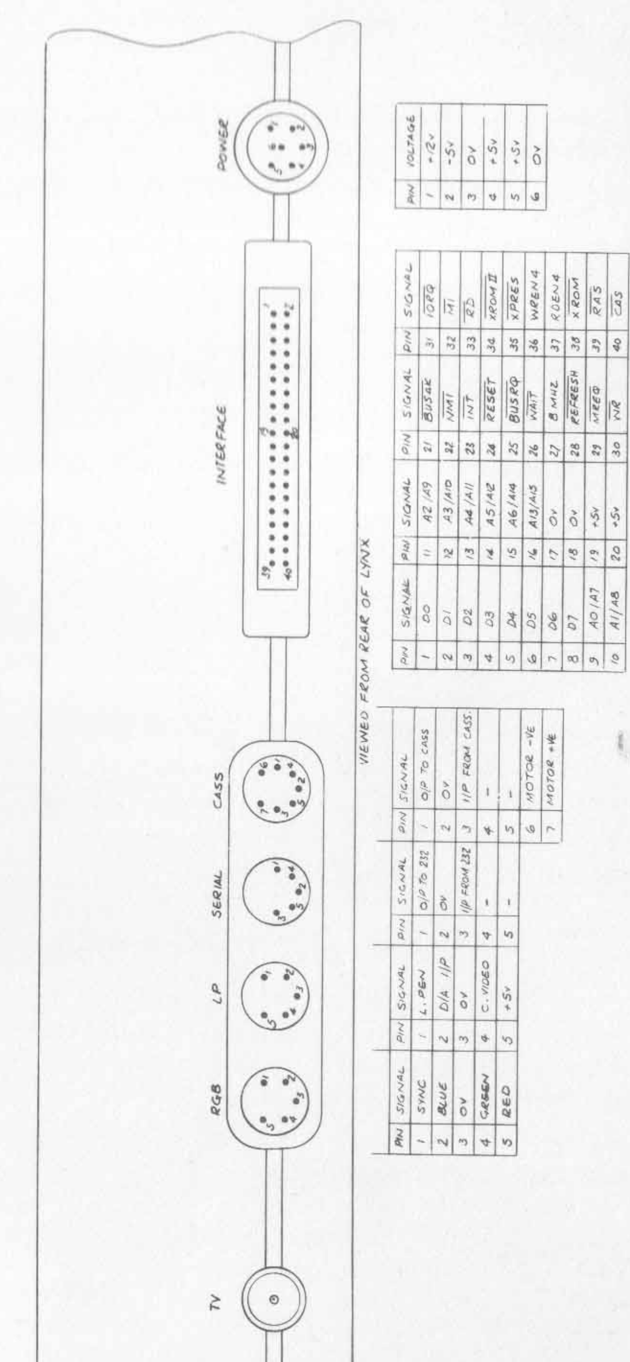
A Auto
B Beep
C Cont
D Del
E data
F deFproc
G Goto
H gosub
I Input
J label
L List
M return
N Next
O endprOc
P Proc
Q rem
R Repeat
S Stop
T Trace
U Until
V Verify
W While
X wend
Y run
Z restore
`PRINT` can be abbreviated to ?

Appendix 3: ASCII CODES

	32	,	44	8	56	D	68
!	33	-	45	9	57	E	69
"	34	.	46	:	58	F	70
#	35	/	47	;	59	G	71
$	36	Ø	48	<	60	H	72
%	37	1	49	=	61	I	73
&	38	2	50	>	62	J	74
'	39	3	51	?	63	K	75
(	40	4	52	@	64	L	76
)	41	5	53	A	65	M	77
*	42	6	54	B	66	N	78
+	43	7	55	C	67	O	79

P	80	\	92	h	104	t	116
Q	81	]	93	i	105	u	117
R	82	©	94	j	106	v	118
S	83	_	95	k	107	w	119
T	84	£	96	l	108	x	120
U	85	a	97	m	109	y	121
V	86	b	98	n	110	z	122
W	87	c	99	o	111	→	123
X	88	d	100	p	112	←	124
Y	89	e	101	q	113	↑	125
Z	90	f	102	r	114	↓	126
[	91	g	103	s	115	⋑	127

Appendix 4: EXTERNAL CONNECTIONS TO THE LYNX



RGB

PIN	SIGNAL
1	SYNC
2	BLUE
3	0V
4	GREEN
5	RED

LP

PIN	SIGNAL
1	L. PEN
2	D/A I/P
3	0V
4	C. VIDEO
5	+5V

SERIAL

PIN	SIGNAL
1	O/P TO 232
2	0V
3	I/P FROM 232
4	–
5	–

CASS

PIN	SIGNAL
1	O/P TO CASS
2	0V
3	I/P FROM CASS
4	–
5	–
6	MOTOR -VE
7	MOTOR +VE

INTERFACE

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
1	D0	11	A2/A9	21	$\overline{BUSAK}$	31	$\overline{IORQ}$
2	D1	12	A3/A10	22	$\overline{NMI}$	32	$\overline{M1}$
3	D2	13	A4/A11	23	$\overline{INT}$	33	$\overline{RD}$
4	D3	14	A5/A12	24	$\overline{RESET}$	34	$\overline{XROM II}$
5	D4	15	A6/A14	25	$\overline{BUSRQ}$	35	XPRES
6	D5	16	A13/A15	26	$\overline{WAIT}$	36	WREN4
7	D6	17	0V	27	8 MHZ	37	RDEN4
8	D7	18	0V	28	$\overline{REFRESH}$	38	XROM
9	A0/A7	19	+5V	29	$\overline{MREQ}$	39	$\overline{RAS}$
10	A1/A8	20	+5V	30	$\overline{WR}$	40	$\overline{CAS}$

POWER

PIN	VOLTAGE
1	+12V
2	-5V
3	0V
4	+5V
5	+5V
6	0V

Appendix 5: SUMMARY OF LYNX BASIC

Anything in *italics* is optional.

Maximum line length is 240 characters.

Variable names are single characters, letters of the alphabet, upper and lower case, 52 variables.

Array names are single characters, letters of the alphabet, upper and lower case, 52 arrays.

`A$ A a A(x) a(x)` can all be used in a single program.

BASIC STATEMENTS

CALL
tells the computer to execute a machine code subroutine at specified address. It is particularly useful with `LCTN` and `CODE`

```
CALL address
CALL LCTN (line number)
```

CCHAR
defines cursor characters

```
CCHAR hex code
```

CFR
sets cursor flash rate

```
CFR number
```

1 (fast) – 65535 (very slow)

CLS
clears the screen and homes the cursor

CODE
stores hex codes as a subroutine

```
CODE code code code.......
```

DATA
stores values to be assigned to variables and string variables by a `READ` statement

```
DATA 100, FRED, e*24.....
```

DEFPROC
marks the beginning of a procedure

```
DEFPROC name
define procedure.......
ENDPROC
```

DIM
dimensions arrays

```
DIM array name (highest subscript)
DIM array, array, array.........
```


DPOKE
loads two adjacent locations with values

```
DPOKE address, value
```

the LSB is loaded into the address, and the MSB into the address+1

END
ends a program

ENDPROC
marks end of procedure

```
DEFPROC
procedure defined
ENDPROC
```

ERROR
generates specified error code

```
ERROR code number
```

EXT
allows for extensions to Basic

FOR TO *STEP* NEXT
sets up a `FOR...NEXT` loop

```
FOR variable name=initial value TO final value
operation
NEXT variable name
```

the counting rate can be set using `STEP`

GOSUB
sends computer to subroutine beginning at specified line

```
GOSUB line number
```

the number can be represented by a variable or an expression

GOSUB LABEL
sends computer to subroutine beginning at line marked by label

```
GOSUB LABEL name......
LABEL name
```

GOTO
sends computer to line specified

```
GOTO line number
```

number can be represented by a variable or an expression

GOTO LABEL
sends computer to line marked by label

```
GOTO LABEL name...
LABEL name
```

IF THEN
allows decision making

```
IF condition THEN operation
```

the operation is carried out only if the condition is true
used with relational and logical operators

IF THEN ELSE
allows decision to be made between two possibilities

```
IF condition THEN first operation
ELSE second operation
```

the operations must be single commands

INPUT
allows response to be typed in

```
INPUT "some text" ; variable name or string variable name
```

`INPUT` provides a `?` prompt

LET
assigns value to variable

```
LET variable name=value
LET variable name=value,variable name = value, ....
```

the value may be represented by a number, a variable or an expression

OUT
sends value to the specified Z80 port
it is an `OUT(C),A` sending argument to `BC`

```
OUT port, value
```

PAUSE

```
PAUSE number
```

10000=1 second approx

POKE
inserts value into specified location

```
POKE address, value
```

PRINT
displays material on the screen

```
PRINT A
PRINT "..some text.."
```

delimiters are `,` – tab and `;` – no space

PRINT@

```
PRINT@ column number, row number; material
```

PRINT TAB

```
PRINT TAB column; material
```

column co-ordinate 0-39

PROC
sends computer to procedure named
can pass parameters

```
PROC name
PROC name (variable name, variable name)
```

RANDOM
reseeds the random number generating function, `RND`

READ
reads values from `DATA` statement

```
READ A,B,C$.......
```

REM
allows REMarks to be inserted into program

```
REM remark..........
```

REPEAT UNTIL
sets up a loop, repeating an operation until some specified condition is fulfilled

```
REPEAT
operation
UNTIL condition
```

the condition is tested at the end of the loop, so the operation is always performed at least once

RESERVE
moves the stack to allow more room for storing machine code

```
RESERVE address
```

RESTORE
restores data pointer either to beginning first line of data or to specified line

```
RESTORE line number
```

the line number can be represented by a variable or an expression

RETURN
marks the end of subroutine, and returns the computer to the line following that containing the `GOSUB`

ROUND ON/OFF
rounds the result of calculation to 6 digits before displaying-otherwise 8 digits displayed
`ROUND` is normally ON

SPEED
sets speed of program execution

STOP
interrupts program as it runs and returns the computer to immediate mode

SWAP
swaps the values of two numeric variables

```
SWAP variable, variable
```

TRACE ON/OFF
displays line number of line about to be executed
`TRACE` is normally OFF

TRAIL ON/OFF
adds trailing zeros to a number to bring it up to an accuracy of 8 digits – or 6 if ROUND is on
`TRAIL` is normally OFF

WHILE WEND
sets up loop to perform an operation whilst a specified condition is true

```
WHILE condition
operation
WEND
```

the condition is tested before the operation is carried out

SYSTEM COMMANDS

AUTO
automatic line numbering

```
AUTO first line number, increment
```

defaults to 100, 10
turn off by pressing *[RETURN]*

CONT
restarts program after *[ESC]*

DEL
deletes part of program

```
DEL line number
DEL first line,last line (inclusive)
```

DISK
calls up the Disk Operating System: lethal if you have no disk drive

LIST
lists program, line, or part of program

```
LIST
LIST line number
LIST first line,last line (inclusive)
```

MON
calls up machine code monitor

NEW
erases program

RENUM
renumbers program

```
RENUM first line number, increment
```

defaults to 100, 10

RUN
runs program

```
RUN line number
```

runs from place specified

EDITING

Edit the last line entered by typing *[CONTROL]Q [RETURN]*.

Edit an earlier line by typing *[CONTROL]E [RETURN]*, then entering line number.

Move the cursor to the left using the left arrow key.

Move the cursor to the right moving the right arrow key.

Move the cursor to the beginning of the program line using the up arrow key.

Move the cursor to the end of the program line using the down arrow key.

Insert by typing in normally.

Delete using the *[DELETE]* key.

Re-enter line by pressing *[RETURN]*.

MATHEMATICAL AND LOGICAL OPERATORS

Grouped in order of algebraic hierarchy, highest first.

symbol	*operation*
**	exponentiation
-	unary minus
*	multiplication
/	division
DIV	integer division
MOD	modulo
+	addition
-	subtraction

>	greater than
<	less than
=	equal to
<>	not equal to
>=	greater than or equal to
<=	less than or equal to
NOT	logical NOT
AND	logical AND
BNAND	16 bit binary AND
OR	logical OR
BNOR	16 bit binary OR
BNXOR	16 bit binary exclusive OR

FUNCTIONS

ABS (X)
gives absolute value of X

ALPHA
returns the (location of ASCII character 32)-320

ANTILOG (X)
gives antilog of X

ARCCOS (cos X)
gives X in radians

ARCSIN (sin X)
gives X in radians

ARCTAN (tan X)
gives X in radians

BIN (X)
tells the computer to treat X (which must be a series of 0s and 1s) as a binary number

BLACK
returns the code number of colour black

BLUE
returns the code number of colour blue

COS (X)
gives cosine of X. X must be in radians

CYAN
returns the code number of colour cyan

DEG (X)
converts X (in radians) to degrees

DPEEK (xxxx)
gives the contents of address xxxx and xxxx+1

EXP (X)
gives e**X

FACT (X)
gives factorial of X

FALSE
returns a value of 0

FRAC (X)
gives fractional part of X

GETN
gives the ASCII code of the key currently pressed; waits until a key is pressed

GRAPHIC
returns the location of character 128

GREEN
returns the code number of colour green

HIMEM
gives the first free address after the stack

HL
gives the value stored in the HL register after the last CALL

INF
returns value as close to infinity as the Lynx is able to process

INK
returns the code number of the current ink colour

INP (port)
gives the value of the specified Z80 port. It gives an IN A,(C) specifying BC

INT (X)
gives integer part of X

KEYN
gives the ASCII code of the key currently pressed; if no key is pressed, gives 0

LCTN (line number)
gives the address in which the first byte of the specified line following the command token is stored

LETTER (code number)
uses `ALPHA` or `GRAPHIC` to return the first byte of the character specified

LOG (X)
gives log of `X`

LN (X)
gives natural log of `X`

MAGENTA
returns the code number of colour magenta

MEM
displays amount of RAM space left in bytes

PAPER
returns the code number of the current paper colour

PEEK (xxxx)
gives the contents of address xxxx

PI
gives value of π, 3.1415927

POS
returns column number of cursor position 0-126

RAD (X)
converts `X` (in degrees) to radians

RAND (X)
returns a random integer between 0 and `X`-1

RED
returns the code number of colour red

RND
generates a random number between 0 and 1

SGN (X)
gives -1 0 or 1, according to the sign of `X`

SIN (X)
gives the sine of `X`. `X` must be in radians

SQR (X)
gives the square root of `X`

TAN (X)
gives the tangent of `X`. `X` must be in radians

TRUE
returns a value of 1

VPOS
returns the vertical position of the cursor 0-240

WHITE
returns the code number of colour white

YELLOW
returns the code number of colour yellow

STRINGS

Valid string variable names are `A$`, `B$`, `C$`,...up to `Z$`.

LET
assigns value to string variable

```
LET string variable name = value
LET A$=... ,B$=...,C$=...
```

the value can be a string (in inverted commas) a string variable, or string expression

DIM
sets maximum length of string up to 127 characters

```
DIM string name (number)
```

anything in excess of specified number is ignored
defaults to 16 characters

CHR$
converts ASCII code to character

```
CHR$ (code number)
```

GET$
returns character string of key pressed
waits for key to be pressed

KEY$
returns character string of key pressed
returns null string if none is pressed

LEFT$
selects left-hand portion of string

```
LEFT$(string name,number of chrs)
```

LEN
gives number of characters in string

```
LEN (string name)
```

MID$
selects middle portion of string

```
MID$ (string name,first chr, no of chrs)
```

RIGHT$
selects right-hand portion of string

```
RIGHT$ (string name, no of chrs)
```

UPC$
converts letters in string to upper case

```
UPC$ (string name)
```

VAL
gives the value of any numbers at the beginning of the string

```
VAL (string name)
```

STRING OPERATORS:

You can concatenate strings using +

You can compare strings using = and NOT:

```
A$=B$                    NOT A$=B$
```

SOUND COMMANDS

BEEP
makes beeping noise

```
BEEP wavelength, no of cycles, volume
```

wavelength 0-65535
no of cycles 0-65535
volume 0-63

SOUND
converts values from specified address onwards into noise

```
SOUND address, delay between outputs
```

delay 0-65535
end marker is a value of 0

GRAPHICS COMMANDS

Screen size is 256 * 247; co-ordinates are specified:

column number,	row number
0-255	0-247

Colours are specified by number or name:

```
0-BLACK
1-BLUE
2-RED
3-MAGENTA
4-GREEN
5-CYAN
6-YELLOW
7-WHITE
```

DRAW
moves cursor and draws a line

```
DRAW co-ordinate, co-ordinate
```

INK
changes 'foreground' colour

```
INK colour number
INK colour name
```

MOVE
moves cursor without drawing line

```
MOVE co-ordinate, co-ordinate
```

PAPER
specifies 'background' colour

```
PAPER colour number
PAPER colour name
```

PLOT
combines MOVE DRAW and DOT in five modes

```
PLOT mode number, co-ordinate, co-ordinate

0=MOVE
1=relative MOVE
2=DRAW
3=relative DRAW
4=DOT
```

WINDOW
sets screen display area
uses text co-ordinates: 40*240

```
WINDOW first column,last column+1, first row, last row+1
```

revert to full screen using

```
WINDOW 3,123,5,245
```

CASSETTE COMMANDS

Program names can be any length, within the maximum line length of 240 characters. They must be in inverted commas.

APPEND
will append a program on cassette to a program already stored in RAM

```
APPEND "name"
```

LOAD
loads a program from cassette into RAM

```
LOAD "name"
```

MLOAD
loads machine code programs

```
MLOAD "name"
```

SAVE
saves a program onto cassette, with optional automatic start

```
SAVE "name"
SAVE "name", line number
```

TAPE
alters baud rate

```
TAPE number 0 (600 baud)-5 (2100 baud)
```

VERIFY
checks that program has been saved successfully

```
VERIFY "name"
```

PRINTER COMMANDS

LINK ON/OFF
links screen and printer displays
`LINK` is normally off

LLIST
lists to printer

LPRINT
prints to printer

MONITOR COMMANDS

Enter monitor by typing `MON` *[RETURN]*. Exit monitor by typing `J` *[RETURN]*.

The monitor has a special prompt `*`, and error message, `????`

`A` arithmetic
`A X Y` displays `X+Y X-Y ZZ` – jump relative

`B` breakpoint
`B X` sets breakpoint at `X`.
`B` restores previous value to `X`.

`C` copy
`C X Y Z` copies from `X` into `Y` for `Z` bytes

`D` dump to cassette
`D X Y Z "name"` dumps from `X` through to `Y` with a transfer address of `Z` and a specified name if transfer address is 0, it will not start up automatically

`E` execute
`E X` runs code from `X`.
`E` runs from stored program counter.

`F` fill
`F X Y Z` fills memory from `X` to `Y` inclusive with byte `Z`.

`G` go
`G X` executes code from `X`, like a subroutine call

`H` hex dump
`H X` dumps contents of memory from `X` to screen as hex and ASCII
`H` dumps from the last `H`, `M`, `L`, `W` etc.

`I` intelligent copy
`I X Y Z` moves block of `Z` bytes from `X` to `Y` intact

`J` jump to Basic

`L` locate
`L X Y` locates occurences of byte `Y`, starting from `X` through to the end of memory abort by pressing *[ESC]*

`M` modify
allows you to modify the contents of RAM
`M X Y1 Y2 Y3 Y4.......`
`M X.`
`M.`
backspace by typing `/` *[RETURN]*.
quit by typing `.` *[RETURN]*.

O output to port
O X Y outputs byte Y to port X.

P increment program counter
P X increments stored program counter by X.

Q query port
Q X displays input from port X..

R read from cassette
R "name" reads file with specified name from cassette

S screen clear

T type into memory
T X text allows you to type ASCII into memory, starting at address X.

U update register
U reg X changes the value of the stored register pair specified to X.

V verify
V X Y Z verifies that the block of Z bytes starting at X is the same as that starting at Y.

W word find
W X Y will search for word Y from X to the end of memory

X display stored contents of registers

Z displays registers and what they point to