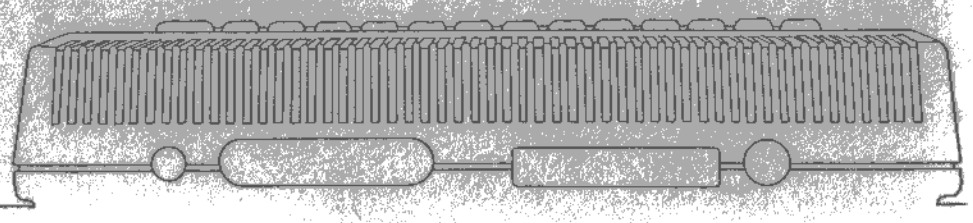
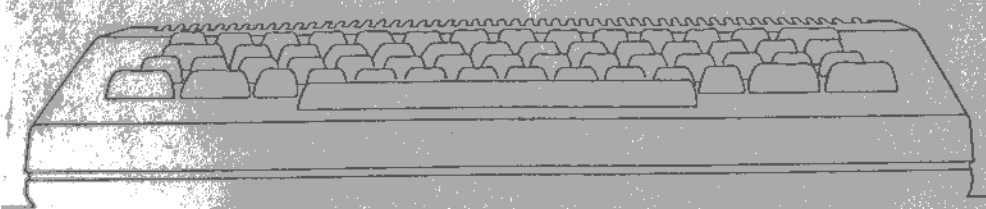
CAMPUTERS

LYNX

128K User Manual





Chapter 1: SETTING UP THE COMPUTER

First examine

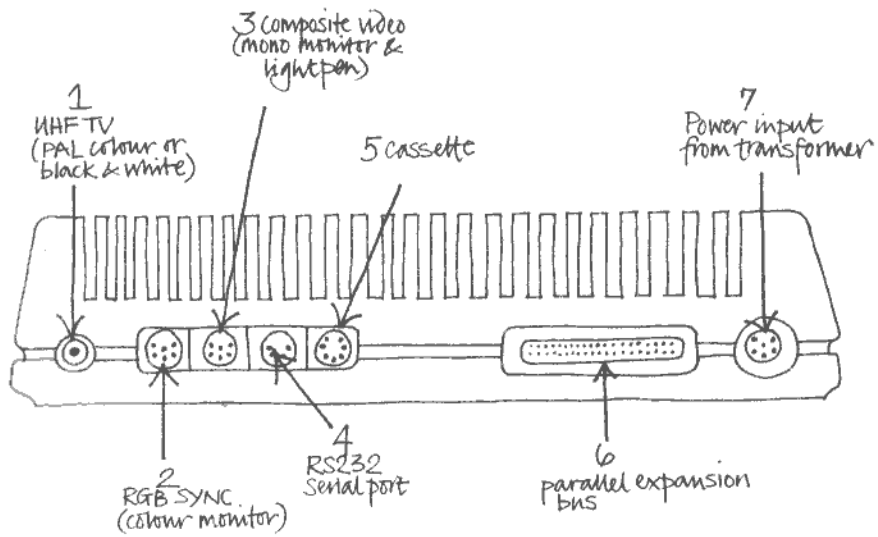
1. The Lynx computer/keyboard, particularly the sockets at the back.

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.

Next 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 succesful. 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 succesful 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 or she is faced with - that is, work out the series of simple steps which bring the answer; then encode this solution into the the language used by the computer. In practice, the two stages are closely linked, because the way programmers solve their problems 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.

This manual begins slowly and simply, but assumes that as you learn more about Basic, you will want to work a litte 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.

COLOURS

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.

TEXT

Another useful command you can try using at this point is TEXT. Just type

TEXT

and press [RETURN]. The Lynx is now in 'TEXT mode'. You'll notice that the screen clears, PAPER turns black, INK turns green and the Lynx seems to work faster. The extra speed can be very useful if you are developing a long program and need to list lots of text and alter it often.

When it's in TEXT mode, the computer can only use green and black -- if you give it any colour commands, the display will be corrupted.

If you want to get out of TEXT mode, type

PROTECT 0 [RETURN].

We'll look at TEXT in more detail -- and see why it makes the machine faster -- in chapter 14.

MEMORY

It may also be useful to know little about the Lynx's memory before you start computing in earnest.

The Lynx has 128K of memory, but not all of this can be used for programming. If you use MEM when you first switch the computer on, the Lynx will display the amount of memory it has available for you -- around 38000 (38K). The computer is using the extra memory itself -- 64K is used to store an internal 'picture' (bit map) of the screen display, about 3K is for 'system use' -- the Lynx uses it as a sort of 'note pad' -- and about 23K is set aside as a Data Store (see Chapter 18).

EXT

Finally, you'll notice that some commands are prefixed with the letters EXT, and may wonder why.

The original 48K Lynx had 16K of ROM (containing the Basic and operating system); but the 128K Lynx supports an extended Basic -- and an extra ROM. It is the commands which are part of this extension which are prefixed by EXT.

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 is similar to [ESC] but is normally ignored by the Lynx: for details see Chapter 19.

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 PI symbol on the keyboard: you enter PI by typing PI.

You can enter an underline character (_) by typing [SHIFT] 0.

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.

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 PI stored in memory with the value of 3.1415927. There is no PI symbol on the keyboard; to use PI 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 statisical 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. And 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 cn restart the progrm 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, because 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.

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, so 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?

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 80 columns -- you can print 80 characters across the screen. These columns are divided into 10 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

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 80 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].

PRINT 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 refering 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 one 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 a 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 a 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 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 in the rest of the program. 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 intruction 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 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$; [RETURN]
30 INPUT ", how old are you";a [RETURN]
40 PRINT "Where do you live, "; A$; [RETURN]
50 INPUT B$ [RETURN]
60 PRINT "And have you lived there ";a;"  years"; [RETURN]
70 INPUT C$ [RTURN]
80 PRINT  "I've never been to ";B$;". What's it like there"; [RETURN]
90 INPUT D$ [RETURN]
100 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.

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

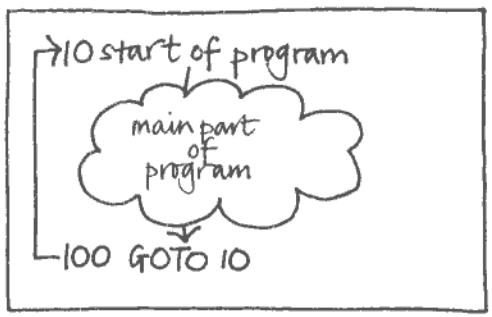
GOTO

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's sometimes tempting to use it to link operations together haphazardly. But this is considered bad programming style because it makes it very difficult for you to see what's happening in your program and modify or improve it.

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.

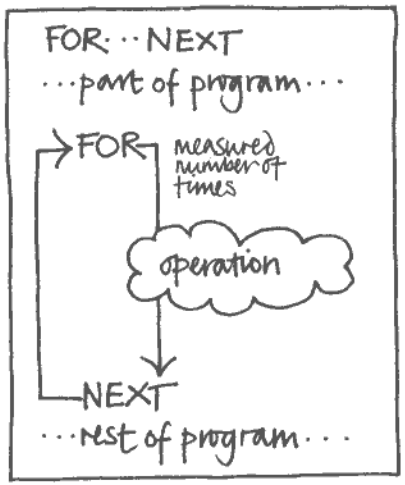
To see this more clearly, try typing in and running this program:

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 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

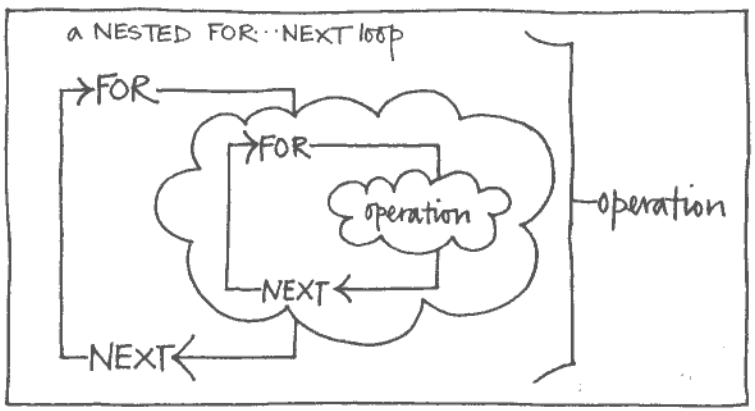
RUN [RETURN]

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 20 to

20 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

1. 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.

They 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 with the = and the > operators. 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 "*"
```

If you compare strings with >, the computer compares them for alphbetical order.

Each of the characters the computer can display, A - Z, 0 - 9, and so on, has a code number (see Chapter 7 and Appendix 3). The computer compares the code

numbers of the characters in each string, starting with the first character. If they are the same it moves to the next, and so on. The higher the code number, the 'greater' the string, so 'B' which has a code of 33 is greater than 'A' with code of 32. 4 (52) is greater than 0 (48);% (37) is greater than $ (36); and

```
"LYNX">"LYMX"
```

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

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

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!"
```

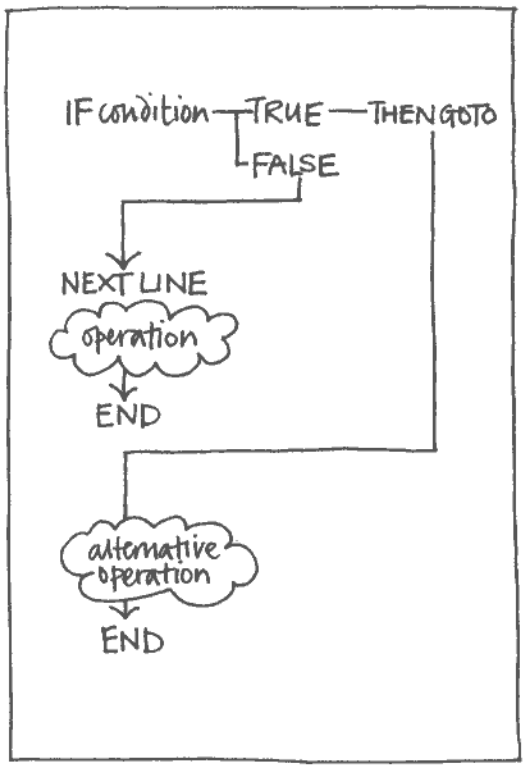
The decision is made in line 20: if your response is LYNX, the computer prints That's my name too!, but not That's a funny name!. Otherwise, it prints That's a funny name!.

Note that the alternatives can only be single commands.

(If, by mistake, you type in an ELSE line without an IF...THEN, 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

1. 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 of 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 cursor (arrow) keys you might do it like this:

```
IF KEYN=12 THEN LET C=C+1
IF KEYN=22 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 12 and 22 are the ASCII codes of the right and left cursor movements respectively. (For more about screen co-ordinates and Control Codes see Chapter 14).

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 16 and 17 on machine code).

STR$

STR$ converts the value of a variable into a string. For example, if

A=7.93

then

STR$(A)

will be the string "7.93"G; if

B=A**2

then

STR$(B)

is "A**2".

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 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(string name)

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]
```

The arguments of VAL, ASC and LEN can be string expressions. For example

```
VAL("7"+A$)

LEN(A$+B$)

ASC("7")
```

IDEAS and EXAMPLES

1. 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.

Note that -- unlike some other computers -- the Lynx doesn't allow you to delete a line with

```
line number [RETURN]
```

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 program, but unlike [ESC], it is a command and 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

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.

SPEED is designed to slow down graphics, so you can see exactly when things start to go wrong. And like TRACE, it 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 17).

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

(holding [CONTROL] down whilst you type Q) 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

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.

Remember that the Lynx does not accept "faulty" lines into its memory. If you type in a line containing a syntax error the Lynx puts it into a temporary storage area and displays an error message, giving you the chance to modify it using [CONTROL] Q. But if you don't notice the error, and type in another line, the temporary copy will be destroyed and you will not be able to recall it -- you'll have to type it in again.

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. (If you have any problems, see Appendix 4 for more information and advice on how to choose a compatible cassette player).

To load and save succesfully it is best to use special computer tapes or high quality audio tapes. Remember that most cassette tapes have a leader, and you will need to wind the tape past this before you can record your program.

Make sure that your cassette player has been installed correctly (see Chapter 1). 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 240 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 "name"

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 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.

Unlike some other computers, if you try to use

LOAD ""

the Lynx will not load the first program it finds -- you must include the program name.

HOW TO ESCAPE FROM SAVE, LOAD AND VREIFY

When you're loading, saving, verifying or appending you can escape at any time using ESC provided there is a signal coming in from the tape recorder.

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

The Lynx can load and save programs at various baud rates. (Baud is the unit in which "data flow" is measured).

You can set the baud rate, using TAPE, like this

TAPE number between 0 and 5

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

The Lynx can LOAD programs at all 6 settings -- a program can only be loaded at the baud rate it was saved at -- and SAVE them at TAPE 3 and TAPE 5.

When it powers up, the Lynx is set to load and save at TAPE 3.

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 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 LET M=0 [RETURN]
60 FOR J=0 TO 12 [RETURN]
70   LETM=M+M(J) [RETURN]
80 NEXT J [RETURN]
90 PRINT "The average mark was ";M/13 [RETURN]
100 REM find the highest mark [RETURN]
110 LET H=M(0) [RETURN]
120 FOR J=0 TO 12 [RETURN]
130   IF M(J)>H THEN LET H=M(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]
```

STRING ARRAYS

You cn also set up arrays of string variables. String array names are single character -- A$(X) to Z$(X) -- and the computer sees A$ (for example) as A$(0), so you can't use both A$ and A$(X) in the same program.

You dimension the array like this

```
DIM array name(number of characters)(highest subscript)
```

so

```
DIM A$(11)(5)
```

sets up an array of 6 elements, and each element can contain up to 11 characters.

You can select any element of the array using

```
Array name(subscript)
```

Here's a very simple program using a string array:

```
100 CLS [RETURN]
110 DIM A$(11)(5) [RETURN]
120 FOR J=0 TO 5 [RETURN]
130  READ A$(J) [RETURN]
140  PRINT A$(J);" COMPUTER" [RETURN]
150 NEXT J [RETURN]
160 DATA LYNX,DRAGON,SPECTRUM,TRS 80,OSBORNE,JUPITER ACE [RETURN]

RUN [RETURN]
```

You can see from line 140 that once data is stored in the array, all the elements can be processed with a single command.

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=2 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$)) [RETURN]
170 LET E$=MID$(D$,4,2) [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, 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 earilier 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 240 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....
```

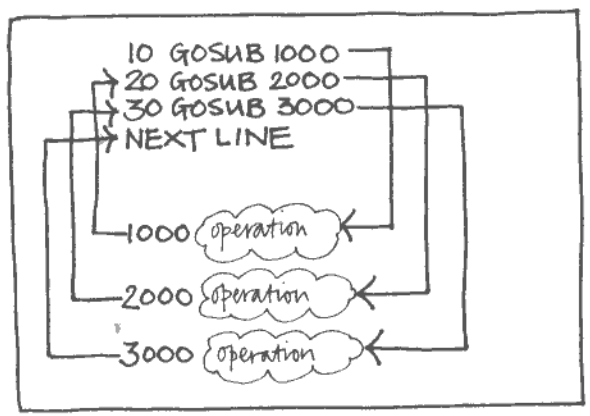
The label has two advantages: first, it is fast. Second, it allows you to write programs which are virtually independent of their line numbers.

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 PRINT "EXECUTING SUBROUTINE..." [RETURN]
180 PAUSE 10000 [RETURN]
190 PRINT "Press a key!" [RETURN]
200 LET G$=GET$ [RETURN]
210 RETURN [RETURN]
220 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.

REPEAT...UNTIL and WHILE...WEND

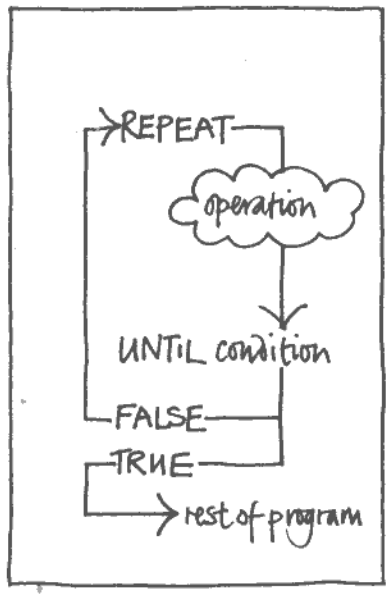
We have already used the FOR...NEXT loop, and seen how versatile it is. Lynx Basic also allows you to use two other types of loop.

REPEAT...UNTIL 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]
```

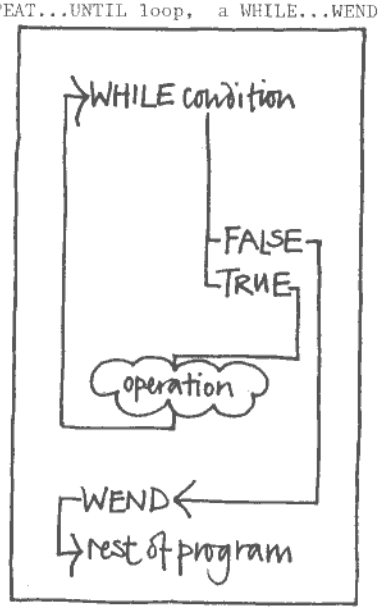
WHILE...WEND

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 varible 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

Printers come in two types: serial and parallel -- the difference is in the way information is formatted and sent to the printer. The Lynx can drive both types of printer. But to use a serial printer you'll need lead; to use a parallel printer you'll need an interface.

When you switch the Lynx, it is set to use a parallel printer; but you can select either mode, using

The Lynx 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

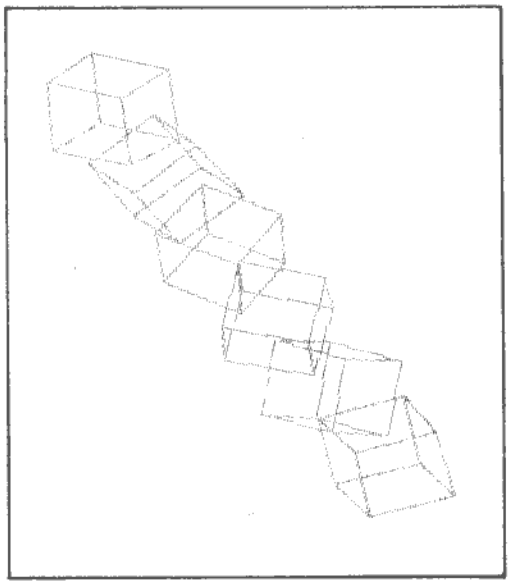
Similarly, LPRINT is the the printer equivalent of PRINT.

LINK

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

For further details, see the Lynx Printer Technical Manual.

Chapter 14: GRAPHICS



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 [RETURN] or PAPER MAGENTA [RETURN] for example.

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

INK colour number

or

INK colour name

All eight colours can be n 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 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 prtect 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 the 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 much faster. So if you protect MAGENTA, you can work in green at approximately twice normal speed. And the Lynx has a command which makes it easy for you to do this -- TEXT.

TEXT

You can put the Lynx into TEXT mode by typing

TEXT [RETURN]

This tells the Lynx to unprotect any colours already protected so that it can clear the screen; to set INK to GREEN and PAPER to BLACK; and then to PROTECT MAGENTA -- disable the RED and BLUE banks -- so that the screen display is speeded up.

You can switch TEXT off with

PROTECT 0

Although TEXT mode is most useful when you are listing and editing programs, you can -- in spite of its name -- use it to speed up graphics as well, but only in green and black. You can alter PAPER to GREEN and INK to BLACK, but you can't use other colours in TEXT mode because the red and blue banks have been disabled.

Try putting the Lynx in TEXT mode then changing the INK colour -- to BLUE, for example. The cursor and prompt will disppear, and nothing you type will appear on the screen. But the cmputer will still be obeying commands, and you can get the display back by typing

PROTECT 0 or INK GREEN

Try experimenting with the fllowing program: it should make PROTECT and TEXT much clearer.

```
100 PROTECT 0 [RETURN]
110 PAPER 0 [RETURN]
120 INK 7 [RETURN]
130 CLS [RETURN]
140 INPUT "Which colour do you want to PROTECT";P
[RETURN]
150 INPUT "What INK clour would you like";I
[RETURN]
160 INPUT "What PAPER colour would you like";B
[RETURN]
170 PRINT "Type a few characters - see what
```



```
effect" [RETURN]
180 PRINT "PROTECT has had."[RETURN]
190 PRINT "PRESS RETURN TO TRY AGAIN" [RETURN]
200 PROTECT P [RETURN]
210 INK I [RETURN]
220 PAPER B [RETURN]
230 INPUT A$ [RETURN]
240 GOTO 100 [RETURN]

RUN [RETURN]
```

If you try to type in more than 240 characters (more than the maximum line length) the computer will ignore the extra ones, but you will still be able to press [RETURN] and carry on experimenting.

You can only escape from this program during the INPUT statement -- hold down [ESC], then press [RETURN].

THE GRAPHICS CHARACTERS

There are 26 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

(holding down the [CONTROL] key whilst you press 1). Then press [SHIFT] and the appropriate key (see diagram).

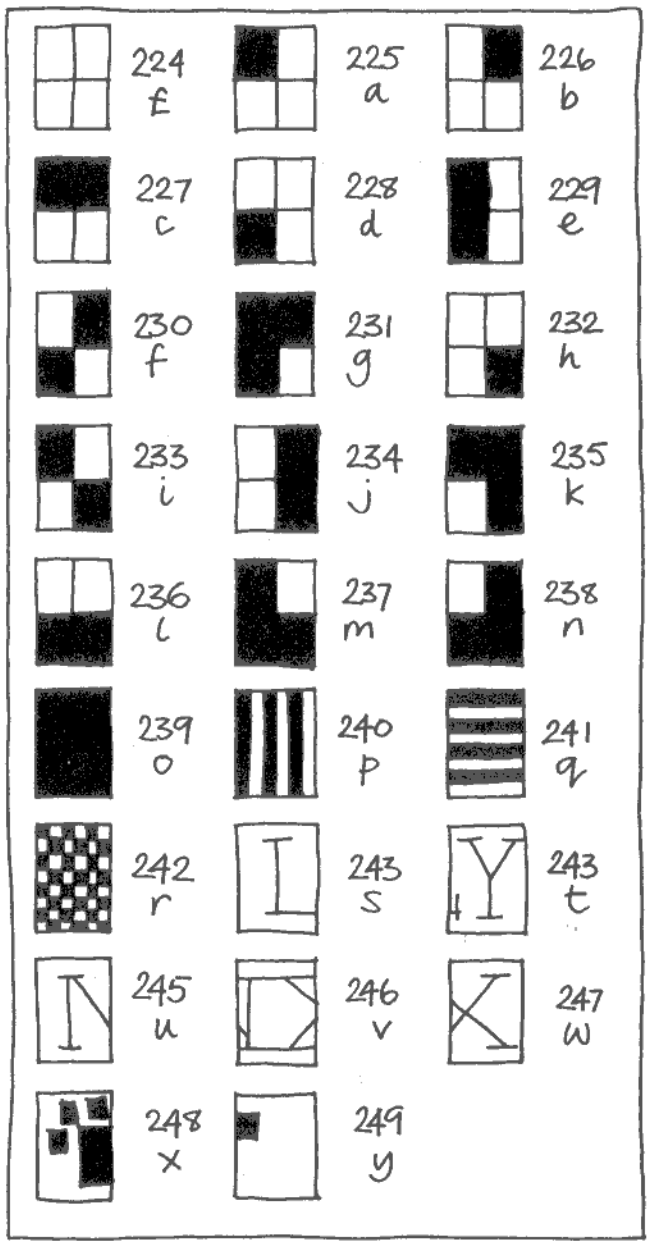
To exit graphics mode, type

[CONTROL] 1

again.

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

```
90 VDU 26 [RETURN]
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]
230 VDU 27 [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 512 * 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 0,1,2 and so on, and the rows similarly. You can refer to each individual dot by its co-ordinate:

column number, row number

But to make planning graphics easy, the Lynx pretends that it can display 512*506 pixels -- you work out your co-ordinates on a square, symmetrical grid, enter them, and then Lynx works out how to fit them onto its screen.

This means that the column number can be between 0 and 511 and the row number between 0 and 503.

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

'LOW RESOLUTION'

The Lynx also has a 'low resolution' mode which allows you to enter co-ordinates as though the screen measured 256*248.

To switch to low resolution mode, type

LOW ON [RETURN]

and to return to high resolution mode type

LOW OFF [RETURN]

In low resolution mode, column number can be between 0 and 255, and row number between 0 and 248. This is the resolution offered by both the 48K and 96K Lynx and this means that graphics programs written on either of those machines -- and loaded from tape or typed in -- can be run on the 128K Lynx.

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 column number, row number

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 column number, row number

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):

```
90 LOW ON [RETURN]
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]
190 LOW OFF [RETURN]

RUN [RETURN]
```

Try changing line 160 to

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


DOT

DOT column number, row number

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:

```
90 LOW ON [RETURN]
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]
160 LOW OFF [RETURN]

RUN [RETURN]
```

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

```
90 LOW ON [RETURN]
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]
170 LOW OFF [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, column number, row number

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.

CIRCLES and TRIANGLES

EXT CIRCLE mode,column number,row number,radius

draws a circle in the current INK colour with its centre at the co-ordinate specified. The mode is selected by a number between -7 and 7, and allows you to draw both filled and unfilled circles and filled and unfilled ellipses which may be "squashed" along the X or Y axis (see table).

You can draw a filled triangle, using

EXT TRIANGLE column no1,row no1,column no2,row no2,column no3,row no3

The co-rdinates set the corners (vertices) of the triangle and must be on screen.

DISPLAYING 40 CHARACTER TEXT

The 128K Lynx normally displays 80 characters across the screen, but it can also display 40 characters. To switch to 40 character mode, type

VDU 26 [RETURN]

and to switch back to 80 characters, type

VDU 27 [RETURN]

(There are a number of very useful VDU commands which we will explore later in this chapter).

WINDOW AND PRINT@

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,

In HIGH RESOLUTION these are:

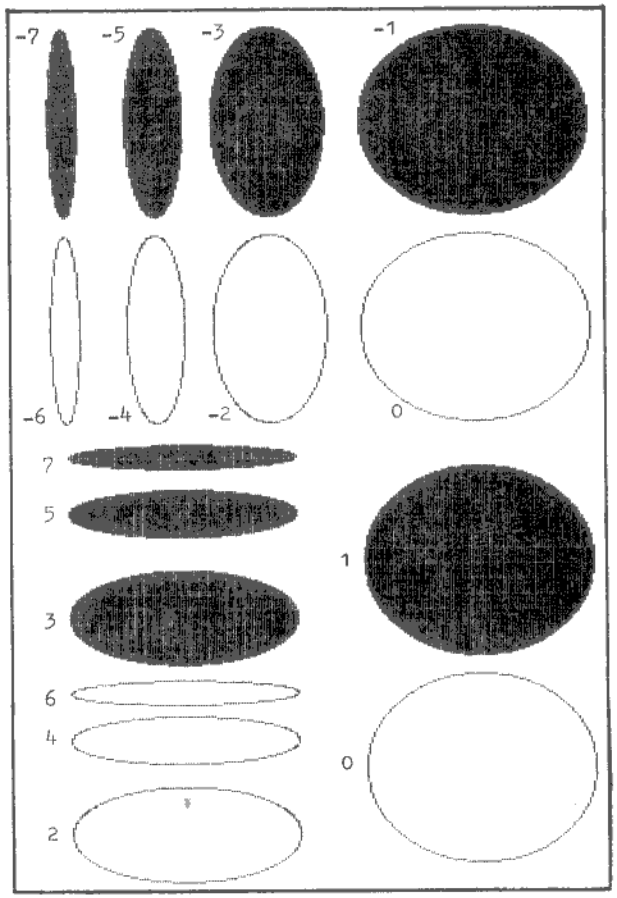
6 - 246 columns

5 - 245 rows

And in LOW RESOLUTION:

3 - 123 columns

5 - 245 rows



WINDOW has this format:

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

The Lynx's normal text window is

6,246,5,245 (3,123,5,245 in LOW RESOLUTION)

so to revert to using the full screen area, use

WINDOW 6,246,5,245 (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@.

You may be wondering why WINDOW has such odd co-ordinates. WINDOW -- and PRINT@ are designed for handling character strings. Characters are made up of a precise number of pixels -- 6 across by 10 down -- and these commands allow for this in the co-ordinate system they use. If you choose co-ordinates which are not a multiple of the pixels in a character block, characters will fall off one side of the screen and reappear on the other side (wraparound).

CLEARING THE WINDOW

EXT CLW

clears the current window with the current paper colour. It can be used as a rectangle fill.

PRINT@

You can use

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

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(240), RAND(240);"*"; [RETURN]
140 GOTO 120 [RETURN]

RUN [RETURN]
```

POS and VPOS

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

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 eg VDU 3,colour number
PROTECTs specified colour

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 eg VDU 11,odd number or even number
odd number on; even number off
displays alternative green (see Chapter 19).

12 moves cursor one character block to the right

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

14 turns internal cursor on (see Chapter 19)

15 turns internal cursor off (see Chapter 19)

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 on

21 overwrite off

22 backspaces cursor

23 homes cursor (takes cursor inside window)

24 turns double height characters on

25 turns double height characters off

26 turns on 40 characters mode

27 turns off 40 characters mode

28 moves cursor up 3 pixel lines (superscript)

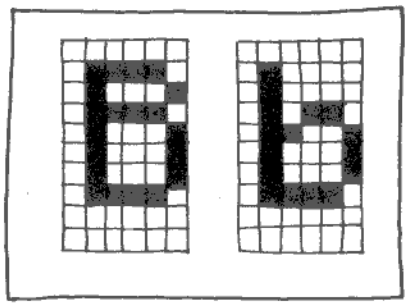
29 moves cursor down 3 pixel lines (subscript)

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.

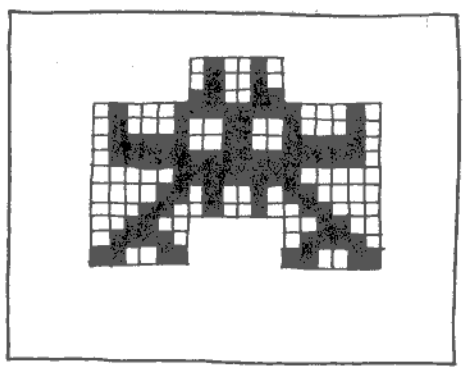
If you take a piece of squared paper, mark out a rectangle of 6x10 squares, then draw a pattern by filling in squares inside the rectangle, you will have designed your own character, which you can then feed into the computer.

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 17). 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 0s and 1s, 1 represents a shaded square, 0 a blank square). BIN, used in line 140, is a function which persuades the computer to treat the 0s and 1s as a binary number.

These numbers are 'poked' (see Chapter 17) 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 making 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.

CHANGING THE CURSOR

Corrupting the Cursor

You may have noticed that by defining graphics characters you seem to have corrupted the cursor.

The Lynx's flashing block cursor is in fact a character block (see Chapter 19 for more details). When you define characters, you shift the pointer to the cursor character away from where the block is stored, and the computer displays whatever is stored at the new position as a cursor.

The standard character set (32 - 127) is not affected by this pointer shift. So you can stop the cursor's metamorphosis by redfining it as one of these characters, using CCHAR.

CCHAR

CCHAR allows you to redefine the 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 a different cursor, using two characters, a character and a space, or, if you want to stop the flashing, the same character twice. And if you want to switch off the cursor you can use two spaces.

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 17) 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
```

Be careful not to define the cursor as a backspace! (If you do, you can regain control by pressing [RETURN] and typing in a new definition).

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.

Corrupting the Graphics Character Blocks

Just as you seem to corrupt the cursor by defining graphics characters so, for the same reason -- shifted pointers -- you also seem to corrupt the pre-defined graphics character blocks.

To get them back again

```
DPEEK (GRAPHIC) [RETURN]
```

when you switch the computer on, and make a note of the number; you can reset the pointer again later, with

```
DPOKE GRAPHIC,value [RETURN]
```

(We will look at DPEEK and DPOKE in Chapter 17).

IDEAS and EXAMPLES

1. Watch this!

```
90 CLS [RETURN]
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: SOUNDS

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 ratio.

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].

The table should help you to convert musical notes into BEEP values, but note that it is not -- strictly speaking -- musically accurate.

SOUND

SOUND address,delay between outputs

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

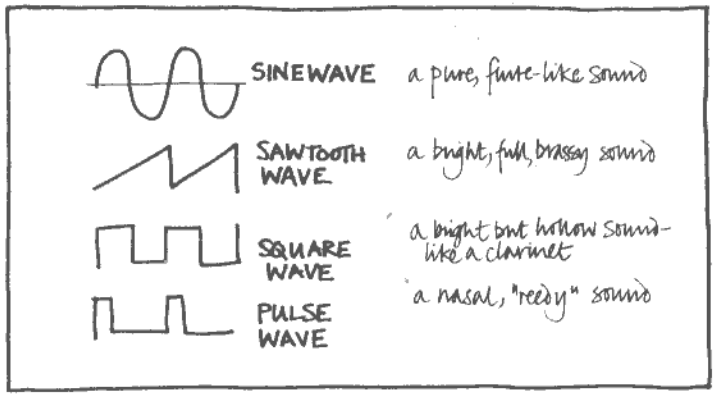
The delay between outputs can be any number between 0 and 65535 and gives the frequency of the sound.

Using RESERVE and HIMEM (see Chapter 17) you can build up sound effects by poking values into memory, then run them using SOUND. Calculating the values you need is a laborious task. To synthesise a sound you need to create a particular shape of wave (SOUND works best with a periodic wave form), then describe that shape with set of values.

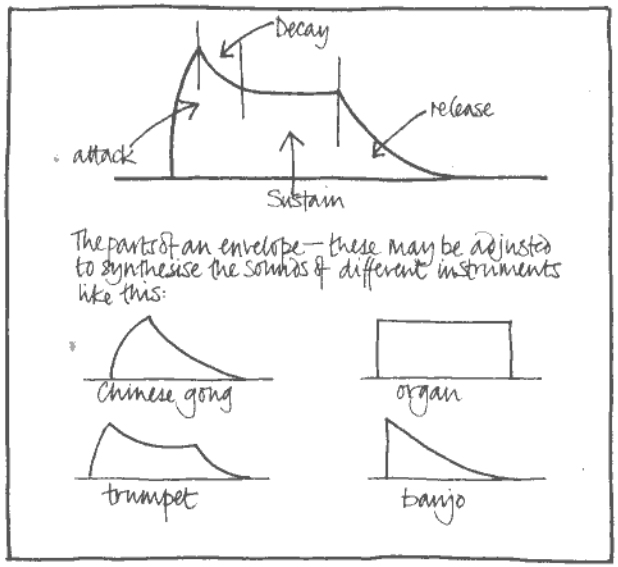
NOTE	FREQ.	W/LEN
A	110	909
A#	116·6	857·3
B	123·5	809·7
C	130·8	764·5
C#	138·6	721·5
D	146·8	681·2
D#	155·6	642·7
E	164·8	606·8
F	174·6	572·7
F#	185	540·5
G	196	510·2
G#	207·7	481·5
A	220	454·5
A#	233·3	428·6
B	246·9	405
MIDC	261·1	383
C	277·2	360·7
D	293·7	340·5
D#	311·1	321·5
E	329·6	303·4
F	349·2	286·4
F#	370	270·3
G	392	255
G#	415·3	240·8
A	440	227·3
A#	466·2	214·5
B	493·9	202·5
C	523·2	191
C#	554·4	180·3
D	587·3	170·3

NOTE	FREQ.	W/LEN
D#	622·3	160·7
E	659·3	151·7
F	698·5	143·2
F#	740	135
G	784	127·5
G#	830·6	120·4
A	880	113·6
A#	932·3	107·3
B	987·8	101·3
C	1046·5	95·5
C#	1108·7	90·2
D	1174·7	85·3
D#	1244·5	80·3
E	1318·5	75·8
F	1396·9	71·6
F#	1480	67·5
G	1568	63·8
G#	1661·2	60·2
A	1760	56·8
A#	18646	53·6
B	1975·5	50·6
C	2093	47·8
C#	2217·5	45
D	2349·5	42·6
D#	2489	40·2
E	2637	37·9
F	27939	35·8
F#	2959·6	33·8
G	31359	31·9
G#	3322·4	30

Compiled by Chris Saffin



This will give you a basic note. To make it sound like a particular instrument you need to "shape" the note as you play it, by altering the amplitude of the waves. The shape of a note is called its envelope and has 4 distinct parts.

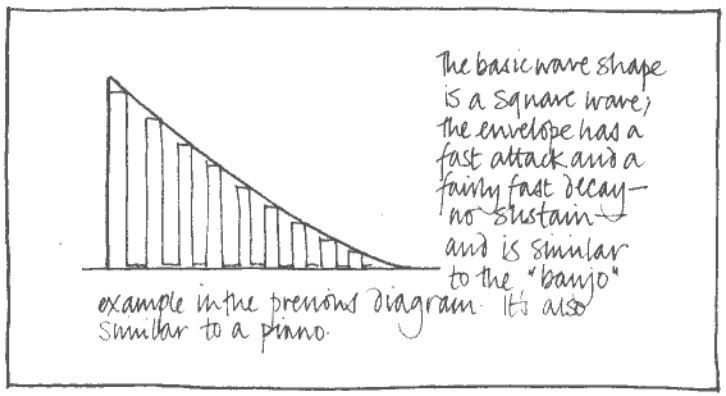


Here's an example:

```
100 RESERVE HIMEM-700 [RETURN]
110 FOR J=0 TO 630 [RETURN]
120  POKE HIMEM+J*2,63-J/10 [RETURN]
130  POKE HIMEM+J*2+1,1 [RETURN]
140 NEXT J
150 INPUT "DELAY BETWEEN OUTPUTS, 100 - 1000";A [RETURN]
160 SOUND HIMEM,A [RETURN]
170 GOTO 150 [RETURN]

RUN [RETURN]
```

The FOR...NEXT loop generates values which create a sound wave like this:



Line 160 "plays" the data from HIMEM onwards, with delay between cycles specified by you in line 150.

ZAP and SPLAT!

The Lynx supports some pre-formatted sounds which you can use in games programs. These can be called by the following commands:

```
EXT LASER
EXT EXPLODE
EXT KLAXON
```

and

```
EXT ZAP
```


Chapter 16: 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 (base 16) numbers.

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

Z80 PROGRAMMING MANUALS

The most popular Z80 programming manual available is

"Programming the Z80" by Rodnay Zaks (published by Sybex).

Another popular book is

"Z80 Assembly Language Programming" by Lance Leventhal (published by Osborne McGraw Hill).

Chapter 17 (11H): 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.

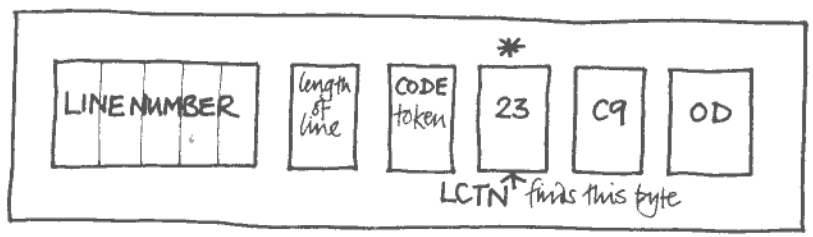
CODE, LCTN CALL and HL

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 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.

Note that CODE lines are not automatically terminated with a RETURN. And that you can freely use any of the registers in CODE line (or in a CALL).

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

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

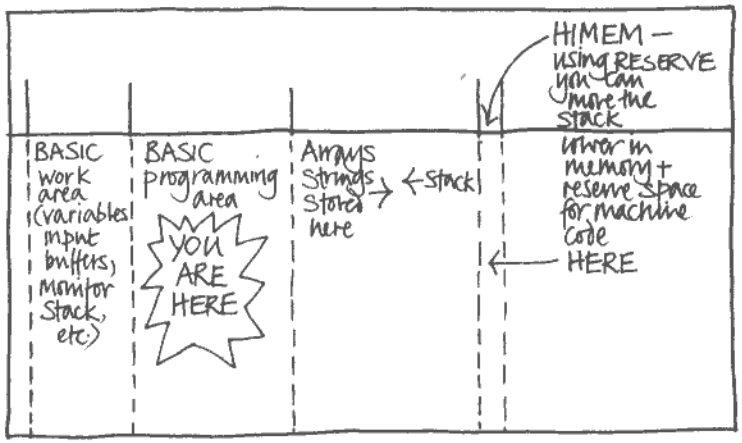
HIMEM is a read-only variable which tells you the first free address after the stack (see the "you are here" map) : it tells you the lowest position in memory from which you can start to store machine code.

RESERVE expression

allows you to move the stack to 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

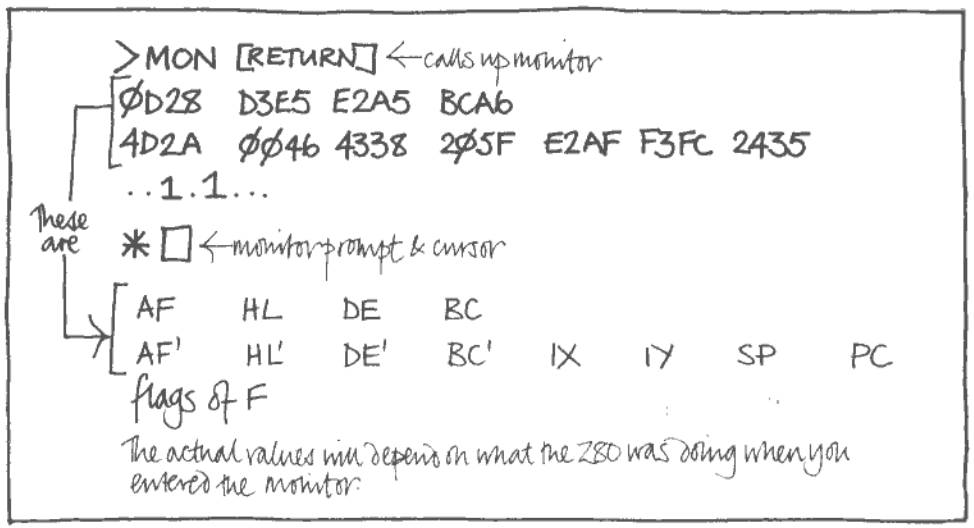
sends a value to a specific port. It performs an

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.

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: when you set a second breakpoint the first one will have its original value 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 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

H X 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 occurences of byte Y, starting at X, through to the end of memory. The address of each occurence 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:

```
*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]
```

N: next (Single Step)

N allows you to execute program one instruction at a time.

N [RETURN]

executes the instruction indicated by the stored Program Counter, then returns to the monitor, which displays the new register values.

You can then alter your program, or you can carry on stepping through it with another P command

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 occurence.

Note that the computer will search for the LSB of Y at the address, and the MSB at the address+1.

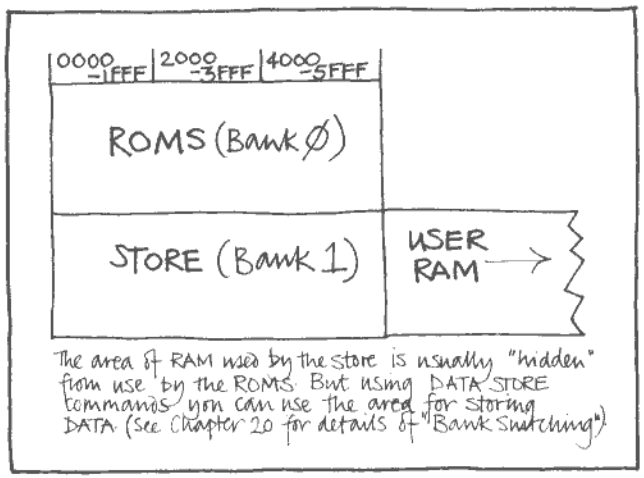
X

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.



The Lynx has 64K of User RAM. About 23K of this memory is set aside as a DATA STORE, which you can use to store data -- numbers, variables, expressions, strings. The Data Store is independent of programs -- you can erase a program with NEW, load another with LOAD, and leave your stored data intact.

When you switch it on, the Lynx sets up the Data Store -- 48 independent stores, labelled 1 to 48, each set to zero.

You can add data to any of the stores at any time, and the stores extend to fill the available space. So, for example, you can use one very large store, filling the whole **K; or one large and several smaller stores -- or any combination. When the computer runs out of storage space, it displays an out of memory error message.

The Lynx keeps track of your data for you using data pointers -- each of the storage areas has its own pointers. As you write a value into a store the 'write' pointer is moved along to the next position -- so if you add more data later it will be stored after the earlier values and they will be left intact. When you read a value back, the computer starts reading from the beginning of the store and shifts the 'read' pointer along to the next value, and the next read starts from this position (unless you use BACK -- see below -- to set the pointer to the beginning of the store again).

DATA STORE COMMANDS

(Remember that EXT is available as a single key entry -- [ESC] E).

STORING DATA

EXT STORE store number,data,data,...

writes the data to the store specified by the number (between 1 and 48).

Data can include numeric variables and expressions -- which are evaluated prior to being stored -- and string variables -- which are stored as the contents of the string in HEX.

String constants must be enclosed by quotation marks.

If the values you want to store are integers between 0 and 65535 these can be stored more economically than other types of data -- saving memory space -- using

EXT ISTORE store number,data,data,...

Storage overhead on different types of DATA

	VALUE	BYTES used
INTEGERS	0 - 15 16 - 4095 4096 - 65535	1 byte 2 bytes 3 bytes
Floating Point		6 bytes
STRINGS		LEN + 1 bytes

MANIPULATING DATA

EXT FETCH store number¦variable,variable....

reads values from the named data store and assigns them to the variables specified, moving the data pointer as it goes. When the data runs out the computer displays an error message.

EXT BACK store number

resets the 'read' data pointer to the start of the named data store -- like RESTORE in Basic.

EXT WIPE store number

erases all data from the specified store by moving the 'write' pointer back to the beginning.

SAVING AND LOADING DATA

EXT SSAVE store number,"name"

saves the contents of the named store to cassette.

When you save a store to tape it loses its number, and is given a name to identify it. The name must be placed in inverted commas and can be any length (within the maximum line length of 240 characters).

Because a data store loses its number when you save it to tape, it can be reloaded into any of the stores.

EXT SLOAD store number "name"

loads the stored data specified by "name" into the data store specified by the number.

CHAINING

EXT CHAIN store number,store number

links together (chains) the two stores specified ready for saving to tape as one continuous data blocks using EXT CSAVE (see below).

EXT CHAIN can only be used when saving to cassette.

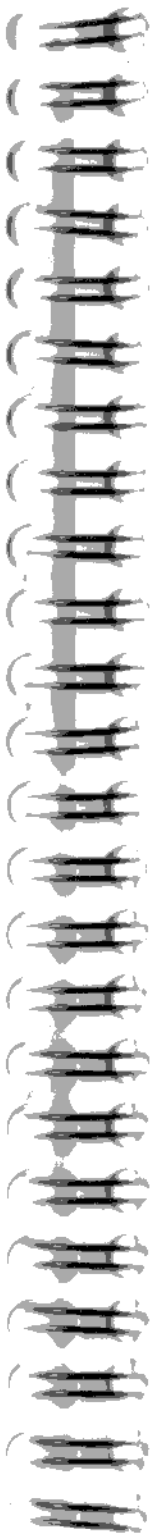
EXT CSAVE number,"name"

saves the two stores chained previously with EXT CHAIN, starting from the store specified. Data saved with this command is reloaded with the usual

EXT SLOAD number,"name"

Here's an example which calculates the values of a sine wave and stores them in Data Store 1 -- in PROC GEN -- then fetches them from the store and displays them on the screen.

```
90 CLS [RETURN]
100 LOW ON [RETURN]
110 PROC GEN [RETURN]
120 PROC DIS [RETURN]
130 PRINT@ 3,240; [RETURN]
135 LOW OFF [RETURN]
```

```
140 END [RETURN]
1000 DEFPROC GEN [RETURN]
1010 EXT WIPE 1 [RETURN]
1020  PRINT  "CALCULATING  VALUES  OF  SINE  WAVE" [RETURN]
1030 FOR X=0 TO 255 [RETURN]
1040 EXT STORE 1, -100*SIN(X/40)+120 [RETURN]
1050 NEXT X [RETURN]
1060 ENDPROC [RETURN]
2000 DEFPROC DIS [RETURN]
2010 CLS [RETURN]
2020 EXT BACK 1 [RETURN]
2030 MOVE 0,120 [RETURN]
2040 FOR X=0 TO 255 [RETURN]
2050 EXT FETCH 1,Y [RETURN]
2060 DRAW X,Y [RETURN]
2070 NEXT X [RETURN]
2080 ENDPROC [RETURN]

RUN [RETURN]
```

ERROR TRAPPING

An error trap is a command which allows you to detect when your program is about to generate an error message, suppress the message, and insert an alternative operation -- a sort of failsafe.

The error trap facility on the Lynx is designed primarily to allow you to detect the end of data store, suppress the ***** error message, and set up an alternative process -- most likely reset the data pointer.

To set an error trap use

EXT TRAP variable name

When the error message is generated it is not displayed; instead its code number is stored in the variable.

You can detect whether the error has occured and set up the alternative operation by testing if the value of the variable is equal to zero in an IF...THEN statement.

Suppressing error messages is quite dangerous -- you can easily crash the machine! You can confine the error trap to a particular part of your program by switching it off with

EXT NO TRAP

The USER functions

The Lynx supports many functions -- ready made sets of instructions for carrying out particular operations. The USER functions work like other functions, but allow you to define the operations they perform.

There are 4 USER functions -- USER0, USER1, USER2 and USER3 -- they take one argument and follow the usual function format:

USER0(value)

If you try to use a USER function without first defining it, the Lynx will display a NOT YET IMPLEMENTED error message.

The USER functions meet the outside world in the area of RAM reserved for system use. This is where you patch your definition into them.

627C	C3 32 3B	FUSERØ:	JP NI
627F	C3 32 3B	FUSER1:	JP NI
6282	C3 32 3B	FUSER2:	JP NI
6285	C3 32 3B	FUSER3:	JP NI

The C3 is a ready-made jump.

As you can see from the table, each USER function jumps out of ROM into RAM and from there is redirected to the NOT YET IMPLEMENTED message which is stored at 3B32H. But if you alter this address to the start address of your routine, the USER function will jump to your definition. Then when you use it, it will perform your operation instead of displying an error message.

Your definition must be in machine code and if it is short and doesn't refer to a specific address inside itself -- is position independent -- it can be written into a CODE line. It can call existing ROM routines.

As an example, let's use USER0 to construct an equivalent to the PEEK function. Peek allows you to examine the contents of a particular address in memory -- the argument specifies the address, the value returned is the contents of the address.

When you use a function the computer stores the argument in an area of memory

called WRA1. The letters stand for Working Register Area. This is an area in RAM which the Basic uses for calculation. The Lynx has several WRAs. The argument is stored in floating point.

There are two useful ROM routines we can use:

FPINT (3497H)

converts the floating point number in WRA1 into an integer in the HL register pair; and

INTFP (34C4H)

which converts the integer in HL to floating point and stores it in WRA1.

Using these two routines, your PEEK equivalent will look like this:

```
USER0     CALL FPINT      ;get argument as integer in HL
          LD L,(HL)       ;get byte in L
          LD H,0          ;set H to 0
          CALL INTFP      ;store HL in WRA1
          RET             ;return
```

You can put this into a CODE line like this:

```
10 CODE CD 97 34 6E 26 00 C3 C4 34
```

then implement it with

```
20 DPOKE &627D,LCTN(10)
```

which makes USER0 jump to the beginning of your routine.

THE CURSOR

Only part of the Lynx screen driver -- the routine in the ROM which controls the screen display -- is used by the Basic.The "cursor" you see -- the flashing block -- is not treated as a cursor internally, and is really just marker to show your position.

But there is a real cursor -- an underline character -- and this is normally switched off.

Try this:

```
10 VDU 14 [RETURN]
20 PAUSE 20000 [RETURN]

RUN [RETURN]
```

An underline will appear at the end of the next line. This is the real cursor, the one which is switched on by VDU 14 and off by VDU 15.

You can use this cursor in your own programs, but the Lynx will automatically

switch it off at the end of the program.

ALTERNATIVE GREEN

The Lynx has 4 areas of memory for storing colours -- RED, BLUE, GREEN and ALTERNATIVE GREEN. It normally uses only three of these areas -- red, blue and green -- but because green and alternative green are parts of the same bank of memory, you can make the Lynx toggle between displaying green and alternative green.

You have to do it in two stages -- first make the computer write to the correct piece of memory, then make it display what you've stored there.

Although it sounds rather mysterious, alternative green is just a 16K block of RAM -- like red, blue or green -- it's the way the Lynx handles it that gives it special characteristics.

You can write to alternative green by selecting

INK 8

(you can't in this case use the colour name).

Then to display what's stored in the alternative green memory, use

VDU 11, odd number

and to go back to displaying ordinary green, use

VDU 11, even number

You can protect alternative green using

PROTECT 8

And the Lynx automatically protects 8 when you switch it on.

The two greens can be used for animation -- you can be modifying one picture whilst you're displaying the other, then display the new picture whilst you're preparing the next.

It can be used for storing a menu, which can be flashed on to the screen at any time without interfering with the main display.

Or, suppose you've written a graphics program which has generated a complicated screen display; then you can examine a list of your program on alternative green without destroying your masterpiece.

To write to alternative green ONLY, first

PROTECT 7

then select either

INK 0 (which will appear black)

or

INK 8 (which will appear green).

RESETTING THE VIDEO

EXT VRESET

resets INK, PAPER, PROTECT, character size and screen format to normal -- power up values. It's useful if you have been experimenting with the 6845 display controller chip (see Chapter 20) nd have altered the screen format.

THE BREAK KEY

When you switch the Lynx on, the [BREAK] key is disabled, but you can enable it by typing

BREAK ON [RETURN]

(and disable it again using BREAK OFF [RETURN]).

If [BREAK] is enabled and you hit

[BREAK] and [ESC]

simultaneously it will reset the computer. Any program will be lost, however.

Chapter 20: LYNX HARDWARE

AN OVERVIEW

The Z80

The Lynx uses a Z80 microprocessor which was originally designed by Zilog Inc., but today it is 'second sourced' by Mostek, Sharp, SGS ATES, and others. It is very popular chip, both for home and business computers, and for industrial control applications because it has a powerful instruction set which includes all the instructions offered by a popular earlier microprocessor, the Intel 8080, and so can run its software, which includes CP/M.

The Lynx makes provision for you to experiment with machine code, both from the familiar world of Basic and through the Machine Code Monitor. But remember that when you program in Basic the operating system protects you from crashing the computer -- by detecting errors and displaying warnings. When you program in machine code the Z80 will obey your instructions without question; if program crashes you will probably have to start again -- by resetting with the [BREAK] key (see Chapter 19) or by switching the Lynx off and on again.

But a crash is not as terrible as it sounds: microprocessors do not know how to read a keyboard, put characters on a screen, and so on, they can only do so by following a precise sequence of instructions. If you interfere with the flow of these instructions, the computer will not be able to carry out normal operations and will seem to crash, but inside, the hardware will still be executing your instructions.

If you want to experiment with machine code, you will need to know some of the principles of Lynx hardware -- so read the rest of this chapter -- and you will need a Z80 programming manual (details of two popular books are given in Chapter 16). Then you can refer to the sections describing CODE statements in Basic programs and the Machine Code Monitor (see Chapter 17).

The 6845 Cathode Ray Tube Controller

The 6845 was originally designed by Motorola, and is now second sourced by Hitachi, Synertek and others. Its function is much simpler than the Z80, since it is essentially a collection of counters. But if, for example, you want to adjust the format of the screen, or alter scrolling, you need to know how it works.

The 6845 can be described as a Programmable Address Sequencer. It 'counts' its way through video memory, from top to bottom, selecting each byte of data in turn and sending it to the TV or monitor (both Cathode Ray Tubes devices) where the Cathode Ray (electron beam) sweeps cross the screen at high speed depositing the data from the video memory onto the screen. The 6845 and the electron beam are synchronised together, with the 6845 in control. For more about the 6845, see later in this Chapter.

High Resolution Video

The Lynx has a high resolution bit mapped colour display. Bit mapped simply means that each point on the screen corresponds to a bit in video memory; more precisely, each point has three primary colours associated with it, red, blue and green, and each of these is either set on or off according to the value of a bit in video memory. Some computers do not have bit mapped screens because many users are satisfied with a text display. Moreover, high resolution takes up a lot of memory. If the computer stores its picture of the screen as ASCII values -- which character number is displayed in which character position -- and generates the picture of the character from special hardware, the screen requires much less memory. But unfortunately, it is very difficult to draw a straight line, a circle, or to redifine characters on this sort of screen. The Lynx is able to display not only text but also sophisticated graphics.

To see how the Lynx handles its screen, let's look at the hardware and software processes involved in displaying a single dot.

The screen memory is arranged with each byte of video memory appearing as a horizontal row of eight bits on the screen. Writing one dot must not change the other seven pixels (bits) in the screen byte, so we need a read-modify-write operation; and this must be done for up to 3 colours. A further complication is that the Z80 can't access the the screen memory whilst the 6845 is accessing it without interfering with the display, so it must be synchronised with the line and frame blanking periods when the electron beam is flying back to start a new line or new frame and normal scanning is suspended.

The procedure for writing a character is equivalent to writing several dots. Accessing the screen via machine code or ROM routines is covered in detail later in this chapter. (See also Chapter 14 for a description of the Basic commands available for controlling the screen).

Bank Switching

The Lynx can be expanded to 192K or 256K of RAM, together with up to 40K of ROM including external ROM. But the Z80 can only address a total of 64K of memory. 'Bank witching' is a technique which allows it to address more -- theoretically, as much as you need. It requires special hardware built around the Z80 and software to drive it. In this section we'll explore the concepts involved in bank switching -- we'll look at the details later!

Normally, in a single bank system, the Z80 is kept busy scanning the keyboard, writing to the screen and executing programs. This also happens in the Lynx, except when you want to access the screen. The screen memory is not in this "normal" bank -- the Lynx screen can use up to 128K, so it has to have 2 dedicated banks.

The video banks are a bit like a parallel world -- you can't peek and poke for them, you have to find the switch that let's you pass through into them. The Z80 can throw the switch by writing to a port. But remember that the Z80 is continually moving through memory in an orderly manner, executing instructions one after another. If you switch banks, all the addresses it manipulates now refer to a different block of physical memory (the parallel world). It is not

directly aware of the bank switch, but it must still recieve the correct sequence of instructions to maintain control of the computer. So program flow must carry on across the boundary: before you switch banks you must place executable code into the target bank plus a mechanism for returning to the normal bank again. You need to be careful!

But once you have this critical code, you can move data between banks and because each bank can be seperately read and write enabled you can be fetching instructions from one bank whilst you are writing to another. You can pass a large block of code to a bank, then execute it by transferring control to that bank when you're ready.

One complication is that is that the ROM bank (bank 0) can be simultaneously read enabled with another bank -- this is ordinarily not possible since the two banks would conflict -- and takes priority over the other bank for all addresses from

0000H to 5FFFH

and also

C000H to DFFFH if XROMI (external ROM) is present

and

E000H to FFFFH if XROMI is present.

This means that Basic need not concern itself with bank switching except when it is accessing the screen. A bank of ROM enabled alone is of little use since even for the smallest programs you usually need some RAM.

Input/Output and System expansion

The Lynx allows for "general purpose" expansion via the forty-way connector at the back. You can attach peripherals such as parallel printer, joystick and disk drives using special expansion packs. These can be ganged together, to a maximum of 3 packs -- limited by the Lynx power supply which is not rated to drive more.

There is provision for up to 16K of external ROM and also a fourth memory bank of up to 64K which would allow the Lynx to run CP/M+ -- an enhanced version of CP/M.

The forty-way connector gives you access to all the main internal bus signals (data, address and control) so you can attach home made hardware -- the electrical details are given later in the chapter,

A number of dedicated I/O signals are provided via the DIN sockets -- full electrical details are given later.

The cassette socket allows you record and replay Basic programs, blocks of memory and blocks of data store. It also provides a high level analogue output signal.

The RGB socket lets you use a colour monitor display (nd on French models provides the signals for a Peritel display).

The serial socket provides serial data channels, both in and out, together with handshake in both directions. It can be used with many different baud rates and transmitting formats, allowing you to communicate with the majority of 'RS232' type devices and other Lynxes (see Chapter 13).

The light pen socket offers 3 useful signals: light pen; a composite video signal, to drive black and white monitor; and an analogue to digital input.

Finally there is a UHF output suitable for running domestic colour or black and white TV. In some countries this may be VHF output.

The Lynx offers several different memory options. The following table summarises their different specifications:

LYNX	USER MEMORY	VIDEO MEMORY	VIDEO RESOLUTION	CP/M CAPABILITY
48K	16K	32K	256x256	NO
64K	32K	32K	256x256	NO
96K	64K	32K	256x256	NO
128K	64K	64K	512x256	YES
192K CP/M+	128K	64K	512x256	YES
192K WITH HI-RES	64K	128K	512x512	YES
256K	128K	128K	512x512	YES

All graphics commands in Basic are compatible with the different screen resolutions.

HARDWARE FEATURES IN DETAIL

WAIT states

WAIT states slow down the operation of the machine. The Z80 is continually executing one machine cycle after another; normally each machine cycle involves a transaction (read or write) between the microprocessor and either memory or an I/O device. Each device (memory or I/O) external to the Z80 has a specified access time. If this time is not met the device may not operate correctly.

WAIT states are inserted by the hardware to lengthen certain machine cycles so that the timings are met. Each machine cycle is measured in 't states' --

there may be 3, 4, 5 or 6 depending on the type of instruction. One t state lasts on sixth of microsecond. A WAIT state is an extra t state added to the normal number.

If you know where t states have been added, and how many, you can work out the length of each machine cycle and time a program precisely.

The following table summarises the rules governing WAIT states. The example shows how to predict where extra t states will occur.

Description		Effect
XROM READ	M1 CYCLES	TWO extra t states
ALL OTHER	M1 CYCLES	ONE extra t state
BANKØ ENABLED MEMORY READ	M2,3,4,5 CYCLES	ONE extra t state
ANY	I/O CYCLES	ONE extra t state
ANY OTHER	CYCLE	NO extra t state

An XROM read cycle is a memory read from bank 0 external ROM -- addresses C000H to FFFFH.

An M1 cycle is any machine cycle where the machine brings the $\overline{M1}$ signal (Z80 pin 27) low -- enables it. This comprises op-code fetches (note that some instructions have a two byte op-code, each byte requiring an M1 cycle) and the interrupt acknowledge cycles (maskable and non-maskable).

An I/O cycle is any machine cycle that reads from or writes to a port. Note that the WAIT STATE added here is additional to the one the Z80 always inserts into I/O operations.

WAIT STATE example

Assume that the bank switch contains 00H nd so both bank 0 ROM and bank 1 RAM are accessible, also suppose that there is external ROM present from E000H to FFFFH.

```
E8FE    LD(HL),A
        M1 fetch opcode             CASE 1
        M2 send A to (HL)           CASE 5

E900    JP (7530)                   ;jump out of XROM into RAM
        M1 fetch opcode             CASE 1
        M2 fetch first address byte CASE 3
        M3 fetch second address byte CASE 3

7530    LD A,
        M1 fetch opcode             CASE 2
        M2 fetch operand            CASE 3

7532    OUT(82),A                   ;switch out bank 0
        M1 fetch opcode             CASE 2
        M2 fetch port address       CASE 3
        M3 output byte to port      CASE 4

7535    RLC(HL)
        M1 fetch first opcode byte  CASE 2
        M1 fetch second opcode byte CASE 2
        M2 fetch (HL)               CASE 5
        M3 send (HL)                CASE 5
```


INTERRUPTS

A Z80 program can be interrupted so that it can respond quickly to unpredictable or infrequent events. There are two types of interrupt -- maskable and non-maskable. Two seperate pins on the Z80 recieve the different signals: $\overline{INT}$ (pin 16)recieves the maskable and $\overline{NMI}$ (pin 17) the non-maskable. These are brought out of the forty-way expansion connector pins 23 (maskable) and 22 (non-maskable). Both interrupts are used by the Lynx itself, you can still use them yourself.

The maskable interrupt pin of the microprocessor can be activated either by the [BREAK] key (for as long as it's pressed) or by the CURSOR signal from the 6845 (pin 19). The CURSOR output is a programmable signal normally intended to generate a visible cursor on a monitor screen, using extra hardware which brightens or inverts the video signal at the appropriate point. But the Lynx uses the signal to provide a simple means of synchronising screen access to the frame blanking period.

When you switch the machine on, the 6845 is initialised to generate the desired picture format and the CURSOR signal is programmed to activate briefly (0.66 us) at the very bottom right hand corner of the screen, just before the beginning of the frame blanking period. This signal is "stretched" so it's easier to detect (40 us +- 8 us) and can be used to divert the microprocessor to updating the screen at a time when the picture will not be corrupted (no snow!).

The computer needs to detect the difference between a CURSOR interrupt and a [BREAK] interrupt. This is possible because the CURSOR signal is brief, whilst the [BREAK] key is held down for a long time (in computer terms).

You can use the maskable interrupt with external equipment since the CURSOR signal can be turned off by reprogramming the 6845 (see the following section). The [BREAK] key cannot be disabled.

NORMAL 6845 REGISTER VALUES		
R0	Horizontal total	5F hex
R1	Horizontal displayed	40 hex
R2	Horizontal sync position	4C hex
R3	Horizontal sync width	37 hex
R4	Vertical total	46 hex
R5	Vertical total adjust	1C hex
R6	Vertical displayed	3F hex
R7	Vertical sync position	44 hex
R8	Interlace & skew	00 hex
R9	Character raster count	03 hex
R10	Cursor start raster address	03 hex
R11	Cursor end raster address	03 hex
R12,13	Start address	00, 40 hex
R14,15	Cursor address	0F, FF hex
R16,17	Light pen address	—

The non-maskable interrupt pin of the Z80 has only one internal use on the Lynx -- single stepping. The single step feature of the Lynx monitor (see Chapter 17) allows you to execute a program one instruction at a time. The hardware, being activated by an input from port 84 causes a non-maskable interrupt an exact number of instructions later, returning control to the monitor, which displays the new register values. When it's not being used for single stepping, the non-maskable interrupt can be used via the forty-way expansion port.

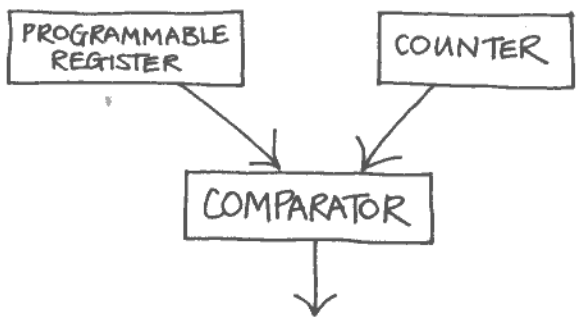
If you want to experiment and substitute your own interrupt routines, it's easy to trace the ordinary program flow to the system interrupt handlers, and redirect it to your own routines, since they're both vectored through RAM calls. Note that you should select only maskable interrupt mode 1, the other two modes are not supported by the hardware.

Programming the Lynx's 6845

The 6845 has seventeen internal registers. When you switch the Lynx on, one of the first things the Operating System Software does is initialise the 6845. If this is not done correctly the picture will not be stable.

The essence of TV or monitor display is timing. The picture is redrawn exactly 50 times a second, each line on the screen is scanned in 64us, each individual pixel is present for one twelfth of a micro second. The display device (TV or monitor) needs information in a video signal to distinguish successive lines of picture data (a line SYNC), and to indicate the end of the whole frame (a frame SYNC). These signals trigger the flyback of the electron beam to the start of a new line or frame. The 6845 provides the majority of these timing signals.

Let's look at some of the internal structures of the IC. For example, Register 1 is the "horizontal displayed" register and works like this: it contains the number of bytes to be displayed along one scan line. Associated with it is a counter register which starts with a value of zero at the beginning of a line, then increments each time a byte is sent to the screen. (This register is 'invisible' to the programmer). The two registers have their contents combined by a device called a comparator which detects when the registers have the same value -- when 32 bytes have been displayed -- and turns the display off.



The other parts of the IC use similar mechanisms: the 6845 is really nothing more than a series of counters. We'll now look at each of its registers in

turn -- read the section as a whole because important points are covered where appropriate. Remember that the 68445 is byte-oriented -- it is not aware of individual pixels, which are clocked out by special hardware.

R0 Horizontal Total

R0 stores the total number of byte periods in one scan.

To work out the value, take the number of bytes displayed, and add the number that could be displayed up to the end of the next line, while the electron beam is turned off, then subtract one from the total. One byte is displayed every 2/3 us; TVs and monitors are designed to have a line scan time of 64 us, so R0 is initialised to 42 decimal or 2A hex. Changing this value is not recommended because it would interfere with correct TV or monitor operation.

R1 Horizontal Displayed

R1 stores the number of displayed byte periods in one horizontal line.

The 128K Lynx normally displays 64 bytes across the screen, so this register is initialised with 64 decimal or 40 hex. Changing the value will affect the width of the screen displayed and disrupt the normal picture, causing a skewing of successive character lines. This skewing happens because -- supposing you reduce R1 to 63, for example -- the character which was at the end of one line now appears at the beginning of the next line, and the effect mounts up as it moves down the screen. Moreover, the skew is based on the 6845s character blocks, which are 4 pixels high (see below).

R2 Horizontal Sync position

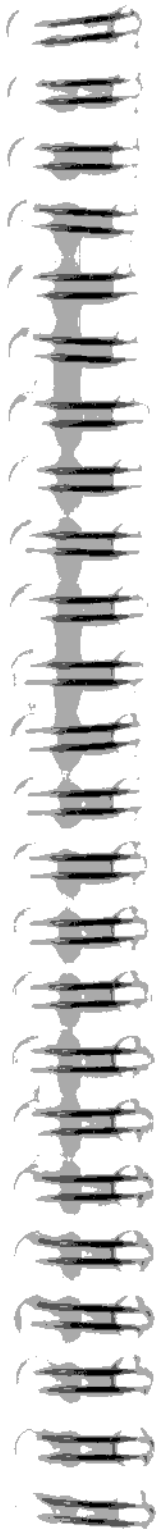
The horizontal sync is used by the TV or monitor to trigger the flyback of the electron beam to the start of the next line. This sync is usually placed roughly central in the line blanking period -- its position will affect the left-right positioning of the picture. Taking the first displayed byte of the line as number zero, the register holds the number of the byte where the sync is located. The 128K Lynx initialises this to 4C hex.

R3 Horizontal Sync Width

The horizontal sync width can be adjusted in terms of byte periods, to match the specification of the TV or monitor involved, and the duration of the byte period. The 128K lynx initialises this register with 37 hex and unless you're using a non-standard monitor you shouldn't need to change it. A value of zero would remove the sync pulse altogether.

R4 Character Row Total

This register is associated with the total number of horizontal scans in one frame, including those which are not displyed. A frame is displayed every 20 ms, and a scan line takes 64 us, so there are roughly 312 scans. But the 6845 is oriented towards character based displays and in this case its characters are set to 4 scan lines high -- the register is initialised with 46 hex -- giving total row count of about 78. This is done because R4 is only 7 bit -- not large enough to store 312 -- so the characters have to be more than 2 pixels high. The value chosen had to be a power of 2 so that the video memory



could be adressed in one continuous block.

A value of (M-1) in the register will give total of M character lines per frame. But you shouldn't need to alter the initilisation value.

R5 Vertical Total Adjust

This register is needed because R4 is too coarse a way of defining the frame duration -- it must be as close to 50Hz as possible. The adjustment is done in scan line units and put into R4. It is initialised at 1C hex, and adjusting it may result in a poor picture.

R6 Character Row Displayed

This register holds the total number of displayed 6845 character rows minus one; the display can only be a multiple of 6845 character rows. The initialisation value of R6 is 3F hex.

R7 Vertical Sync Position

The vertical sync is used by the TV or monitor to trigger the flyback of the electron beam to the top of the screen. It is programmed in multiples of 6845 character rows (the required number minus one) and is initialised to 44 hex which positions the picture centrally on the screen.

R8 Interlace Mode

This is a 2 bit register which selects mode operation and on the 128K Lynx it is initialised to select a non-interlaced display.

An interlaced display is a means of producing a high resolution picture without losing speed, using up too much memory or needing a very expensive monitor. To produce a display with 512 horizontal lines, alternate frames display alternate sets of 256 lines -- in a particular frame only every other line of the picture is displayed.

R9 Maximum Raster Address

The 6845 uses characters 4 pixels high but this has no effect on the height of the characters actually displayed since they are under software control. R9 sets the number of scan lines in a character row -- the pixel height of the characters. It is initialised to 3 (one less than the number of scan lines needed. Any adjustments to this register really need to be supported by hardware changes to achieve anything useful.

R10 Cursor Start Raster and R11 Cursor End Raster

The 6845 provides the signal to generate a "hardware cursor" on a display. The Lynx's flashing block cursor is generated by software; and the hardware cursor is used to interrupt the Z80 at the very end of the visible screen, and indicate that frame blanking is about to begin. R10 and R11 select which rows, or bytes, of the 6845 character the cursor is covering generate the cursor signal. The Lynx needs only one interrupt per screen, at the very base of the screen, so both R10 and R11 are set to 3. The upper 3 bits of R10 control

cursor blinking, but do not apply, so bit 6 is set to zero. If bit 5 is set, the cursor signal will disappear altogether; so you can use 23H to disable cursor interrupts.

R12 and R13 Start Address

This 14 bit register determines the memory address from which the first byte on the screen is fetched. Scrolling can be controlled via this register; but there is no control over the raster start address, and verticl scrolling is by four pixel increments.

Strictly speaking, this register holds a character address (we'll look at the actual memory address a character occupies later, when we consider the memory map of the video system). The 6845 -- with its "characters" -- was originally designed for use in low resolution displays using a character ROM rather than high resolution display.

Character addresses increase from left to right, and from top to bottom, in much the same way as the screen is scanned, except that a character row contains 4 scan lines. Adding or subtracting 64 from the start address causes a one character line scroll (4 pixels). You can obtain an imperfect form of horizontal scroll by using other values.

R14 and R15 Cursor Position

The cursor position register holds the address of 6845 character on the screen (see R10 and R11 for the special use made of the cursor). The cursor is placed at the bottom right hand corner of the screen on initialisation; R14 is loaded with 0F hex and R15 with FF hex.

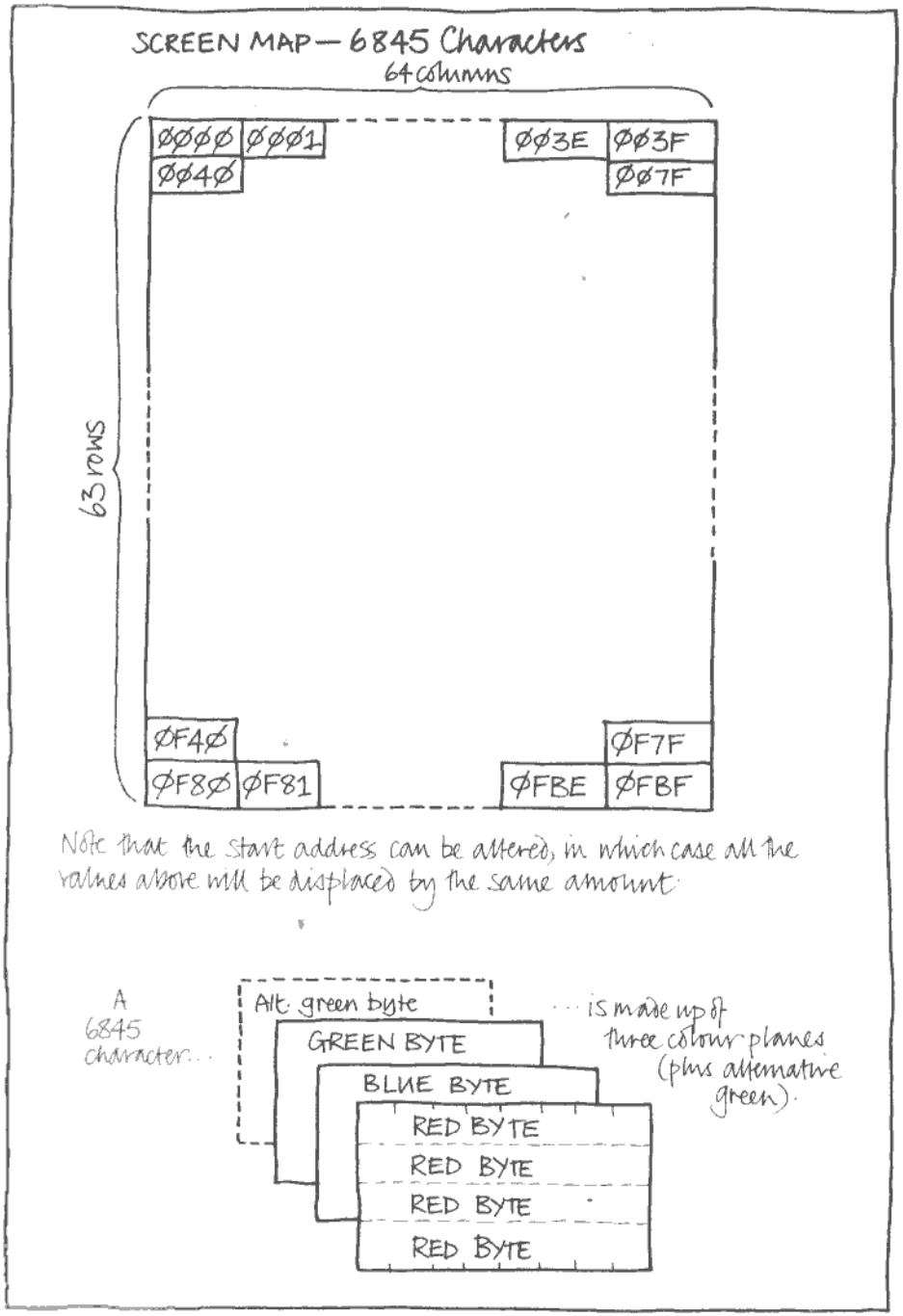
R16 and R17 Light Pen Position

The light pen position register is a read only register which will contain the character address of a light pen, providing the necessary electrical connections have been made. Note that this limits the accuracy of a light pen to a rectangle 8 pixels x 4 pixels, although this can be improved with clever software!

Driving a Bit Mapped Screen

The 128K Lynx has 64K bytes of memory for the display and 64K for User RAM. These occupy two seperate banks and you need to control the bank switch to access video memory. In this section we'll look at the memory map and some examples of how to access the screen using machine code.

In the previous section we saw that the 6845 has characters which are 8 pixels wide and 4 pixels deep. The following diagram illustrates how these chracters cover the screen in 63 rows and 64 columns, and how each of the characters has three colour planes, blue, red and green (together with alternative green, which can be displayed instead of ordinary green -- for more details see Chapter 19.



The screen is further subdivided into individual byte addresses, each of the four colour blocks (red, blue, green and alternative green) taking up 16K each. The following diagram shows how these addresses are mapped to the physical screen. Note that if you change the 6845 start address, this will change.

Read or write accesses to the screen should be synchronised to either the line blanking or the frame blanking period. Line blanking lasts for about 22 us at the end of each visible line and can be used for single byte transfers; frame blanking lasts much longer -- about 3.4 ms -- but occurs only once every 20 ms, so is suited to larger transfers of data. Remember that if you don't synchronise screen accesses you'll get brief 'blackouts' -- tiny black lines -- on the part of the picture currently being drawn.

The following examples show how you can read and write bytes during interline blanking. The first example executes code entirely in user RAM. The interline access bit of port 80 is used to make the Z80 wait until the next frame blanking period; this is invisible to the programmer, and is achieved by "bus requesting" the CPU, to suspend operation. The CPU access bit of port 80 is used to give access to the video memory.

WRITING A BYTE TO THE SCREEN

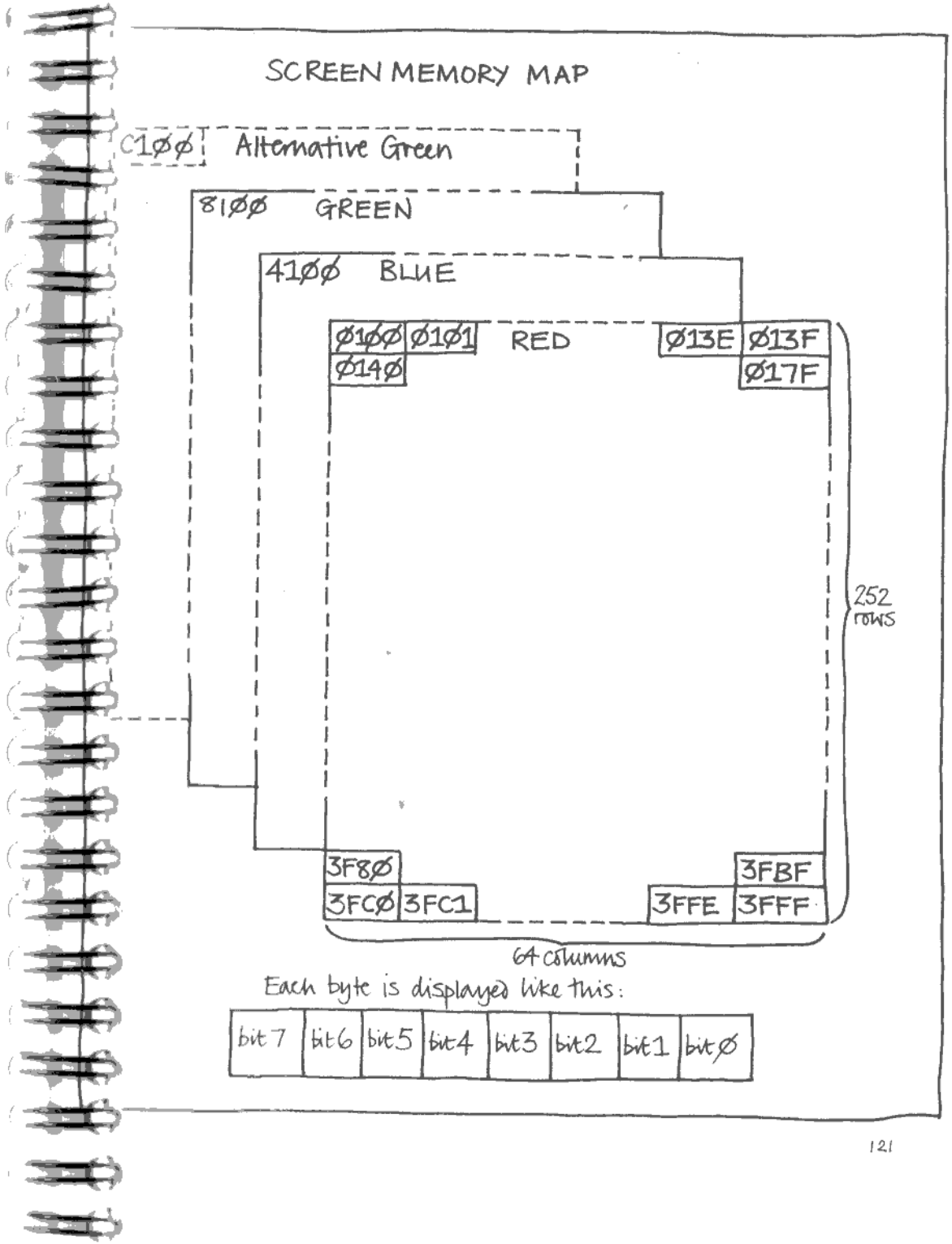
```
8000     21 AB 16  LD HL,16ABH       ;HL points into video RAM
8003     16 7B     LD D,7BH          ;D holds byte to be written
8005     3E C0     LD A,11000000B
8007     D3 82     OUT (82H),A       ;enable write to bank 2, disable bank 1
8009     3E 40     LD A,11000000B    ;A holds mask to set interline bit
800B     D3 80     OUT  (80H),A      ;CPU waits here until interline blanking
800D     3E 20     LD A,00100000B    ;A holds mask to set CPU access bit
800F     D3 80     OUT (80H),A       ;CPU has access to video
8010     72        LD (HL),D         ;write byte to video
8011     AF        XOR A             ;clear A register
8012     D3 80     OUT (80H),A       ;disable CPU access
8014     D3 82     OUT (82H),A       ;reset bank latch
8016     C9        RET
```

The second example is more complex because to read a value from the screen you also have to be reading op-codes (only one bank can be read enabled at a time).

The first part copies 5 bytes of code across into the video RAM using the write routine above. Interline synchronisation and CPU access is achieved as before, but the bank switch is now done after these operations because it's only after you've achieved synchronistion and access that you can fetch op-codes from bank 2 video RAM.

READING A BYTE FROM THE SCREEN
(uses write byte routine)

```
8030     21 4E 80  LD HL,804EH       ;HL points to code to be moved to
                                      video RAM
8033     06 05     LD B,5H           ;B is loop counter
8035     56        LD D,(HL)         ;D holds byte of code
```

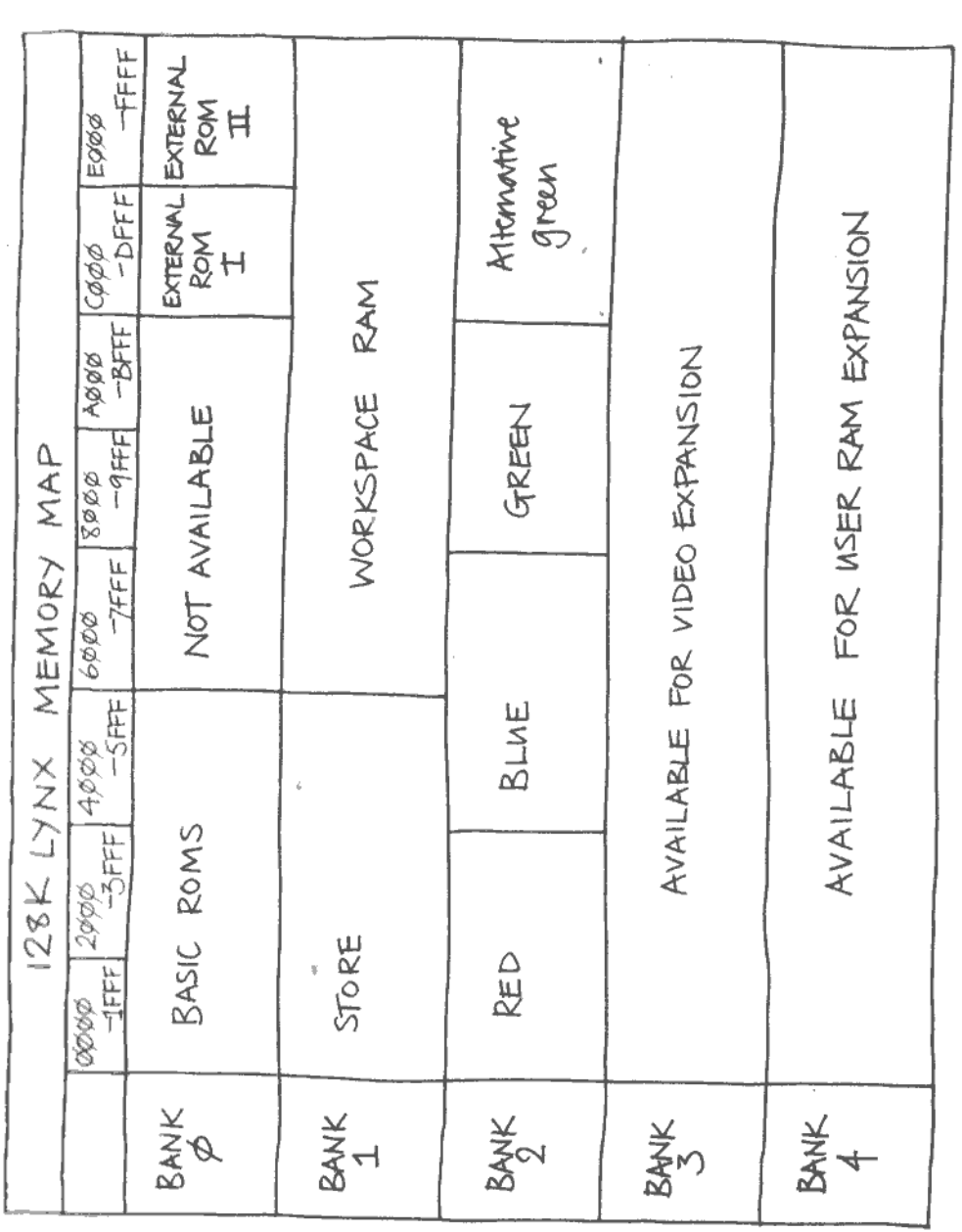



```
8036    CD 05 80   CALL (8005H)        ;call the write byte routine
8039    23         INC HL
803A    10 F9      DJNZ F9H            ;loop to copy five bytes
803C    01 60 80   LD BC,8060H         ;BC is destination address in user
                                        RAM
803F    21 4E 80   LD HL,804EH         ;HL is source address in video
8042    3E 40      LD A,01010000B
8044    D3 80      OUT (80H),A         ;set CPU access bit
8046    3E 20      LD A,00100000B
8048    D3 80      OUT (80H),A         ;set CPU access bit
804A    3E 0E      LD A,00001110B
804C    D3 82      OUT (82H),A         ;enable read from video, write to
                                        user, now fetching opcodes from
                                        video
804E    7E         LD A,(HL)           ;read byte from video
804F    02         LD (BC),A           ;write byte to user RAM
8050    AF         XOR A               ;clear A
8051    D3 82      OUT (82H),A         ;restore bank latch; now fetching
                                        opcodes from user RAM
8053    D3 80      OUT (80H),A         ;reset CPU access
8055    C9         RET                 ;return
```

BANK ARCHITECTURE AND CONTROL

The 128K Lynx's bank architecture is shown in the diagram.

Bank 0 is read only and contains the Basic ROMs. Bank 1 is roughly divided as shown for Basic, or else can be turned over completely to CP/M. Note that banks 3 and 4 are available for expansion of video and user RAM respectively.



128K LYNX INTERNAL I/O

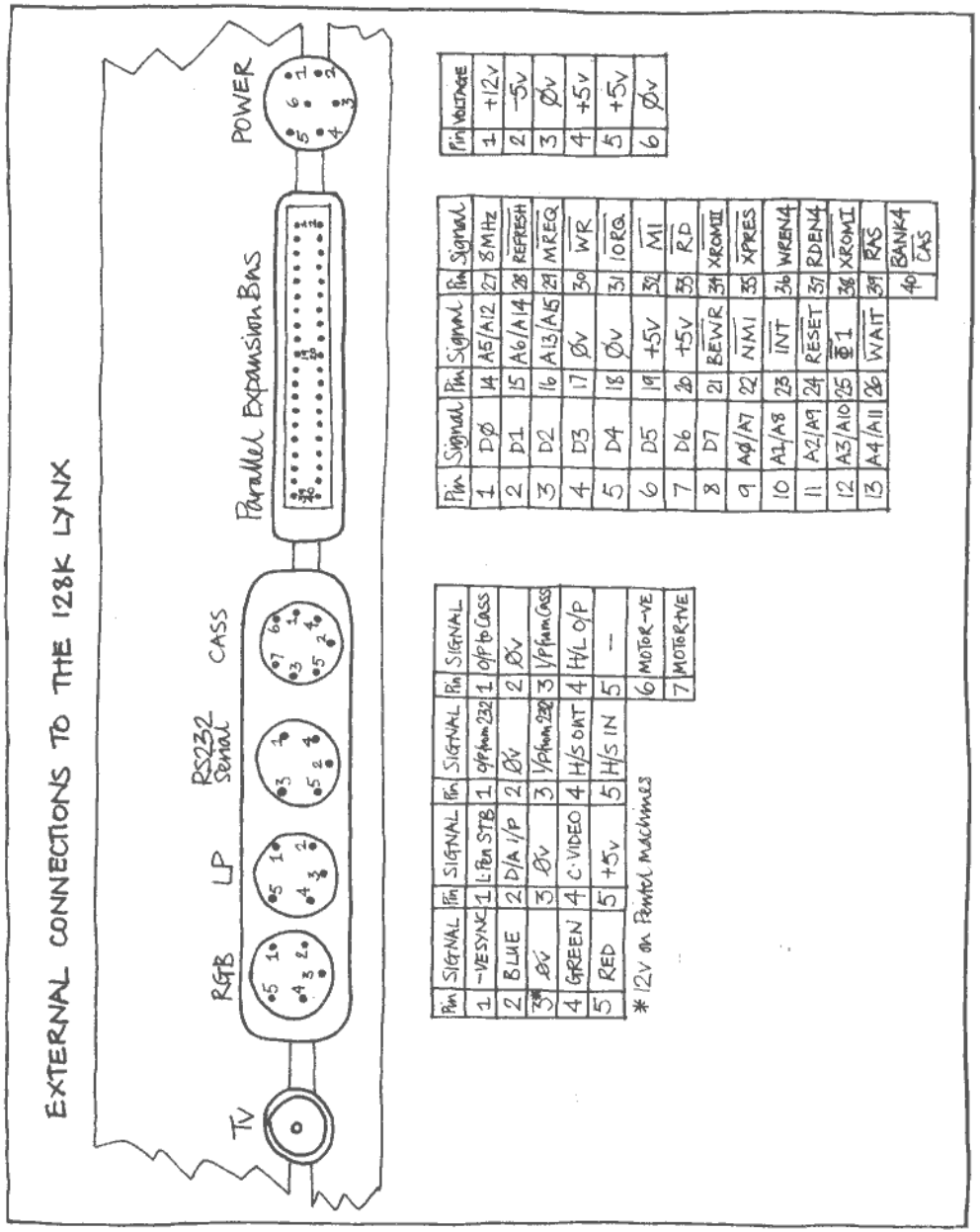
FUNCTION	DATA D7	D6	D5	D4	D3	D2	D1	DØ	ADDRESS 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 Ø	RD/LR	conventional address
CRTC WRITE	← REGISTER DATA →								1 Ø Ø Ø 1 1 1 1	OUT	87
	← REGISTER SELECT →								1 Ø Ø Ø Ø 1 1 Ø	OUT	86
D/A LATCH			B5 MSB	B4	B3	B2	B1	BØ LSB	1 Ø Ø Ø Ø 1 Ø Ø	OUT	84
BANK LATCH	WREN1	WREN2	COLSEL	WREN4	RDEN3	RDEN1	RDEN2	RDEN4	1 Ø Ø Ø Ø Ø 1 Ø	OUT	82
VIDEO LATCH	SERIAL DATA OUT	INTERLINE CONTROL ACTIVE HIGH	CPU ACCESS ACTIVE HIGH	GRN/ ALT GRN 1=ALT GRN	CASSETTE MOTOR ACTIVE HIGH	CASSETTE ENABLE ACTIVE HIGH	SERIAL H/S OUT	SPEAKER ACTIVE HIGH	1 Ø Ø Ø Ø Ø Ø Ø	OUT	80
CRTC READ	← DATA FROM READABLE REGISTER DEFINED BY PREVIOUS REGISTER SELECT →								1 Ø Ø Ø 1 1 1 1	IN	87
SINGLE STEP									1 Ø Ø Ø Ø 1 Ø Ø	IN	84
SERIAL BUFFER	CLOCK					CASS ANALOGUE IN	SERIAL DATA IN	SERIAL H/S IN	1 Ø Ø Ø Ø Ø 1 Ø	IN	82
KEYBOARD	SHIFT	ESCAPE	↓	↑	SHIFT LOCK			1	Ø Ø Ø Ø 1 Ø Ø Ø	IN	Ø080
			C	D	X	E	4	3	Ø Ø Ø 1 1 Ø Ø Ø	IN	Ø180
		CONTROL	A	S	Z	W	Q	2	Ø Ø 1 Ø 1 Ø Ø Ø	IN	Ø28Ø
			F	G	V	T	R	5	Ø Ø 1 1 1 Ø Ø Ø	IN	Ø38Ø
			B	N	SPACE	H	Y	6	Ø 1 Ø Ø 1 Ø Ø Ø	IN	Ø48Ø
			J		M	U	8	7	Ø 1 Ø 1 1 Ø Ø Ø	IN	Ø58Ø
			K		,	O	I	9	Ø 1 1 Ø 1 Ø Ø Ø	IN	Ø68Ø
			;		.	L	P	Ø	Ø 1 1 1 1 Ø Ø Ø	IN	Ø78Ø
			:		/	[	@	−	1 Ø Ø Ø 1 Ø Ø Ø	IN	Ø88Ø
			→		RETURN	←	7	DELETE	1 Ø Ø 1 1 Ø Ø Ø	IN	Ø98Ø

LYNX EXTERNAL I/O MAP

DESCRIPTION	DATA D7	D6	D5	D4	D3	D2	D1	DØ	I/O ADDRESS A7	A6	A5	A4	A3	A2	A1	AØ	DATA DIRECTION	
DISK	1793 RØ−R3								*	1	Ø	1	Ø	Ø	ØØ to 11		READ	50-53
	1793 RØ−R3								*	1	Ø	1	Ø	1	ØØ to 11		WRITE	54-57
	1=125ns Ø=250ns	1=No PRECOMP	NC	EPROM ENABLE	MOTOR	SIDE	SEL 1−4		*	1	Ø	1	1	Ø	Ø	Ø	WRITE	58
JOYSTICK	SPARE	SPARE	FIRE	SPARE	RIGHT	LEFT	DOWN	UP	*	1	1	1	1	Ø	1	Ø	READ	~~7A~~
	SPARE	SPARE	FIRE	SPARE	RIGHT	LEFT	DOWN	UP	*	1	1	1	1	Ø	1	1	READ	~~7B~~
CENTRONICS PRINTER INTERFACE	PRINTER STATUS WORD					SEL	PE	BUSY	*	1	1	1	1	1	Ø	Ø	READ	7C
	PRINTER INITIALISE SIGNAL								*	1	1	1	1	1	Ø	1	R/W	7D
	DATA 8	DATA 7	DATA 6	DATA 5	DATA 4	DATA 3	DATA 2	DATA 1	*	1	1	1	1	1	1	Ø	WRITE	7E

* = "Don't care"

NOTE When a port is specified as write, then a read from that port address will also effect a write, and vice versa, a write to a read port results in data being put onto the bus.



Pin	Voltage
1	+12v
2	-5v
3	0v
4	+5v
5	+5v
6	0v

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	D0	14	A5/A12	27	8MHz		
2	D1	15	A6/A14	28	REFRESH		
3	D2	16	A13/A15	29	MREQ		
4	D3	17	0v	30	WR		
5	D4	18	0v	31	IORQ		
6	D5	19	+5v	32	M1		
7	D6	20	+5v	33	RD		
8	D7	21	BEWR	34	XROMII		
9	A0/A7	22	NMI	35	XPRES		
10	A1/A8	23	INT	36	WREN4		
11	A2/A9	24	RESET	37	RDEN4		
12	A3/A10	25	Ø1	38	XROMI		
13	A4/A11	26	WAIT	39	RAS	40	BANK4 CAS

Pin	SIGNAL	Pin	SIGNAL	Pin	SIGNAL	Pin	SIGNAL
1	-VESYNC	1	L.Pen STB	1	o/p from 232	1	o/p to CASS
2	BLUE	2	D/A i/p	2	0v	2	0v
3	0v	3	0v	3	i/p from 232	3	i/p from CASS
4	GREEN	4	C.VIDEO	4	H/S OUT	4	H/L o/p
5	RED	5	+5v	5	H/S IN	5	--
						6	MOTOR-VE
						7	MOTOR+VE

* 12v on Peritel machines

RGB

PIN 1 -VE SYNC 100Ω impedance, TTL levels

PINS 2,4,5 BLUE, GREEN, RED, 100Ω TTL levels

PIN 3 0v standard
12v Peritel

SHROUD 0v

LP

PIN 1 LIGHT PEN STROBE TTL input with 10K pull up to +5v
requires -VE strobe

PIN 2 Analogue to Digital Input
Measurement range 0 to 2.9V
Input impedance >1MΩ
6 bit accuracy

Time averaging should be used to remove noise.

Uses same input as cassette, therefore disconnect cassette and enable input using bit 2 of port 80.

Level should be found by successive approximation using port 84 D/A and interogating bit 2 of port 82.

PIN 3 OV

PIN 4 Composite video
1V peak to peak
0.3V sync
0.7V video (+ve going)
75Ω impedance

SHROUD OV

SERIAL

PIN 1 SERIAL DATA OUT HIGH LEVEL 11V MIN @ 10 mA
LOW LEVEL -3V MAX @ -1.6 mA

PIN 2 OV

PIN 3 SERIAL DATA IN INPUT IMPEDANCE >10K

PIN 4 HANDSHAKE OUT HIGH LEVEL 11V MIN @ 10 mA
LOW LEVEL -3V MAX @ -1.6 mA

PIN 5 HANDSHAKE IN INPUT IMPEDANCE >10K

SHROUD CASSETTE — 0V

PIN 1 — CASSETTE OUTPUT, low level D/A output with appropriate filtering

PIN 2 — 0V

PIN 3 — CASSETTE INPUT, connects to analogue to digital input (LIGHT PEN socket, pin 2) via 47nf capacitor.

PIN 4 — High level D/A output range 0 to 2.9V
6 bit accuracy
output impedance < 1KΩ

PIN 5 — NOT CONNECTED

PIN 6,7 — CASSETTE MOTOR RELAY 24V 1A
Not suitable for switching mains

SHROUD — 0V

EXPANSION CONNECTOR

PINS 1 - 8 — D0 - D7 buffered Z80 data bus

PINS 9 - 16 — A0/A7 - A13/A15 multiplexed Z80 address bus.
The address lines are multiplexed in order to drive dynamic memories. Multiplexing only occurs when MREQ is active and RFSH is not. Hence I/o cycles do not involve address multiplexing, and only addresses

A0 A1 A2 A3 A4 A5 A6 A13

will be accessible. These addresses are stable on the falling edges of $\overline{MREQ}$, $\overline{RD}$ and $\overline{WR}$ which are available on the interface; the other set are stable within 100 ns of the falling edges. However, hobbyists need not demultiplex the address bus for I/O devices.

PINS 17,18 — 0V

PIN 21 — $\overline{BEWR}$ - Buffered early write. Modified Z80 write signal only active for memory write transactions, falling edge occurs before Bank 4 $\overline{CAS}$ falling edge, but after $\overline{RAS}$ falling edge.

PIN 22 — $\overline{NMI}$ - Z80 $\overline{NMI}$ signal, 1K5Ω pull up internally

PIN 23 — $\overline{INT}$ - Z80 $\overline{INT}$ signal, 470Ω pull up internally

PIN 24 — $\overline{RESET}$ - Z80 $\overline{RESET}$ signal, 470Ω pull up internally

PIN 25 — Φ1 - Z80 CLOCK

PIN 26 — $\overline{WAIT}$ - Z80 $\overline{WAIT}$ signal, 680Ω pull up internally

PIN 27 — 8 MHz clock

PIN 28 — $\overline{REFRESH}$
PIN 29 — $\overline{MREQ}$
PIN 30 — $\overline{WR}$
PIN 31 — $\overline{IORQ}$
PIN 32 — $\overline{MI}$
PIN 33 — $\overline{RD}$
} BUFFERED Z80 signals

PIN 34 — $\overline{XROMII}$ - active on a bank read, addresses E000 to FFFF

PIN 35 — $\overline{XPRES}$ - input to indicate that externl ROM is present, hence used as a "handshake" for $\overline{XROMI}$ and $\overline{XROMII}$

PIN 36 — $\overline{WREN4}$ - Bank 4 write enable

PIN 37 — $\overline{RDEN4}$ - Bank 4 read enable

PIN 38 — $\overline{XROMI}$ - Active on a bank 0 read, addresses C000 to DFFF

PIN 39 — $\overline{RAS}$ - Row address strobe for external dynamic memories

PIN 40 — BANK4 $\overline{CAS}$ - Column address strobe for bank 4 dynamic memories.

Appendix 1: The 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 explaination.

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.

Instead 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 Esc
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 ?

	32	,	44	8	56	D	68	P	80	\	92	h	104	t	116
!	33	-	45	9	57	E	69	Q	81	]	93	i	105	u	117
"	34	.	46	:	58	F	70	R	82	©	94	j	106	v	118
#	35	/	47	;	59	G	71	S	83	_	95	k	107	w	119
$	36	∅	48	<	60	H	72	T	84	£	96	l	108	x	120
%	37	1	49	=	61	I	73	U	85	a	97	m	109	y	121
&	38	2	50	>	62	J	74	V	86	b	98	n	110	z	122
'	39	3	51	?	63	K	75	W	87	c	99	o	111	→	123
(	40	4	52	@	64	L	76	X	88	d	100	p	112	←	124
)	41	5	53	A	65	M	77	Y	89	e	101	q	113	↑	125
*	42	6	54	B	66	N	78	Z	90	f	102	r	114	↓	126
+	43	7	55	C	67	O	79	[	91	g	103	s	115	☺	127

Appendix 4: CASSETTE RECORDERS

The Lynx is designed to work with wide range of cassette recorders. But if you own one which performs badly recording or playing music, don't try to use it for recording programs.

CLEANING

To make sure that your recordings are good and that the signal is clear when you play it back, you need to clean tape heads and the pinch roller regularly, especially after playing a new tape for the first time, and tape heads should be demagnetised regularly -- you can buy cleaning tapes and demagnetising tapes from most shops which stock cassette players.

MAKING UP A HI FI LEAD

The standard lead supplied with the Lynx will fit most mono recorders. If you have a Hi-Fi recorder, you will have to make up a special lead -- use the MIC and EAR equivalents; LINE IN/OUT or DIN levels may not be suitable. On stereo units, it is best to use only one channel for replay rather than a mono output made by mixing right and left channels together. For wiring details of the Lynx DIN connector, see Chapter 20.

THE STANDARD CASSETTE LEAD

The colours of the Lynx lead may vary according to our suppliers.

MIC has a lighter coloured lead or plug
EAR is usually black
REM has the narrowest plug

These are connected to three sockets on the cassette player:

MIC -- where the signal from the Lynx enters.

EAR (or an equivalent word, eg speaker) -- where the signal comes out.

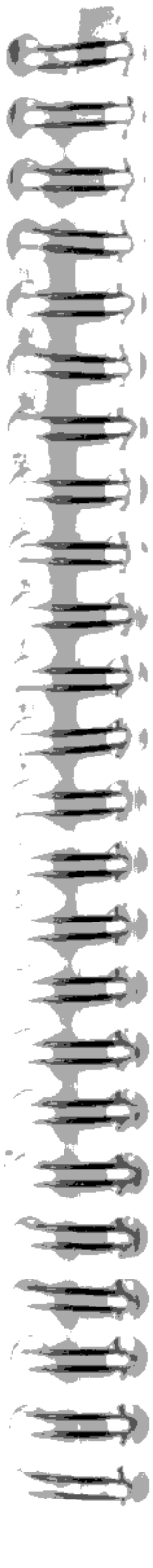
REM the remote control -- the Lynx will function without this.

HAVING PROBLEMS?

1 Check you're using the right volume setting -- see Chpater 1 -- and once you've found it, don't forget to make note of it!

2 If you're having problems loading and saving, disconnect your cassette plyer from the Lynx and test it like this:

Without any tape in the recorder, PLAY at maximum volume. Any hissing sound should be barely audible, and if there is noticeable low frequency hum, try using fresh internal batteries only -- no mains lead!



To test the MIC socket, you should record some music from undistorted radio or record player through a microphone plugged into the MIC socket, then play back. The sound should not be distorted with the volume set at 7 (3/4 of the range).

To test the EAR socket, insert an earpiece or headphones into EAR and listen whilst moving the EAR plug about slightly in the socket: there should not be any crackling sounds or breaks in the music.

Test REM like this. Connect the REM plug into the REM socket but leave the other plugs loose. Insert a music cassette into the player, press PLAY and type SAVE "X" [RETURN]. If the music starts playing and sounds normal, and stops when the prompt appears, the remote control is suitable.

If your cassette recorder has an internal microphone it should automatically disconnect it when you plug the Lynx lead into the MIC socket. If it does not, your programs will be corrupted by the extra noises recorded.

To test this, connect MIC and EAR, but not REM, press PLAY and RECORD and talk into the internal microphone. Then rewind the tape and play it back. If you can hear your voice then the internal mike is not being disconnected and the recorder is not suitable for recording programs.

To check the reliability of your tape recorder's speed control, pinch wheels, and so on, record a dummy program (see Chapter 1). Then rewind the tape, disconnect the EAR lead and play the program back.

You will first hear a continuous tone: if this is distorted, or varies in pitch or volume in any way, your recorder may not be suitable for recording programs.

The Lynx will accept minor variations, but cannot cope with breaks or wobbles in the signal.

Some cassette players have their EAR socket wiring reversed. To test if this is the problem, disconnect the Lynx leads from the player, insert an earpice or headphones, and listen to any recorded tape. Take a short piece of wire -- an unfolded paper clip is ideal -- hold one end firmly against the outer metal ring of the EAR socket and touch the other end to the similar outer ring of the MIC socket. If the sound volume is reduced or stops when you do this, then the wiring of your cassette player is reversed, and the machine is not compatible with a computer.

Written and illustrated by Sue Jansons