Appendix II

An *introducion* to the background of computing

Whose afraid of the jargon ?

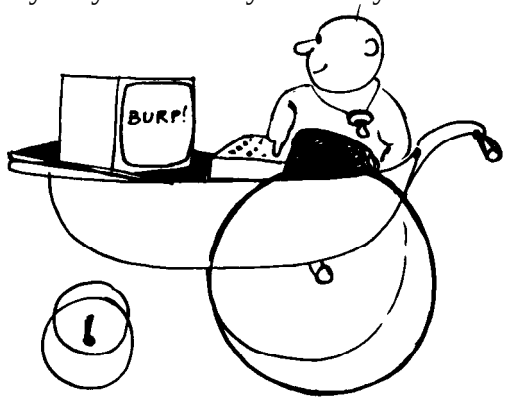
As with all 'specialist' industries, computing has developed its own jargon as a short-hand form of communicating complicated concepts that require many words of 'plain language' explanation. It's not just the high technology business that's guilty of hiding itself behind an apparent smokescreen of 'buzz words', jargon and terminolgy - most of us have come up against the barriers to understanding, erected by all the main professions and trades.

A major difference is that the confusion in legal jargon arises from the way the words are used - rather than the words themselves, as is the case with computing. Most people who grow familiar with computing terminology will go out of their way to use the words in the most straightforward possible manner, so as to minimize the complexity of the communication.

Don't be mislead by the 'plain language' used in computing, it is not a literary subject, but a precise science, and apart from the 'syntax' of the wording, the structure of the communication is very straightforward, and not in the least confusing or ambiguous. Teachers of computing have not yet managed to make an art form out of trying to analyse the exact meaning intended by a programmer in his program construction.

Having said that, although whether or not the meaning of a computer program is obvious, there are still many aspects that can be analysed as either elegant or untidy, and more emphasis is being put on a formal approach to program construction now that the initial mayhem brought about by the micro revolution is settling down.

Computing is rapidly understood by many young people who appreciate the precision and simplicity of the ideas and the way they can be communicated - you don't find too many 10 year old lawyers - but you can find plenty of ten year old programmers !



Basics of BASIC

Virtually all home computers provide a language known as BASIC, which allows programs to be written in the nearest thing to plain language presently available. BASIC is an acronym of 'Beginners All-purpose Symbolic Instruction Code - it no longer has any particular significance as to the degree of the sophistication of the languages, and many extremely complex and powerful programs are written using BASIC.

However, there's no doubt that the name has attracted many newcomers for its promise of providing a starting place in the maze of computer program languages, and this has contributed significantly to its universality.

From here onwards, the commonly used words that form the glossary of terms of computing will be introduced by first printing them in italics. Don't worry about trying to learn them from the following sections - there's an index to them in the glossary.

Basics

BASIC is a computer language that interprets a range of permitted commands, and then performs operations on data while the program runs. Unlike the average human vocabulary of 5-8 thousand words (plus all the different ways verbs can be used etc.), BASIC has to get by with about two hundred. Computer programs written using BASIC have to follow rigid rules concerning the use of these words. The syntax is precise, and any attempt to communicate with the computer using literal or colloquial expressions (plain language) will result in the cold and clinical message:

Syntax *error*

This is not as restrictive as it first appears, since the language of BASIC (the Syntax) is primarily designed to manipulate numbers - the, numeric data. The words are essentially an extension of the familiar mathematical operators + / -etc. - and the most import concept for newcomers to grasp is the fact that a computer can only work with numeric data - information that is supplied to the ***Central*** Processor integrated circuit is only supplied in the form of numerical data.

NUMBER PLEASE!

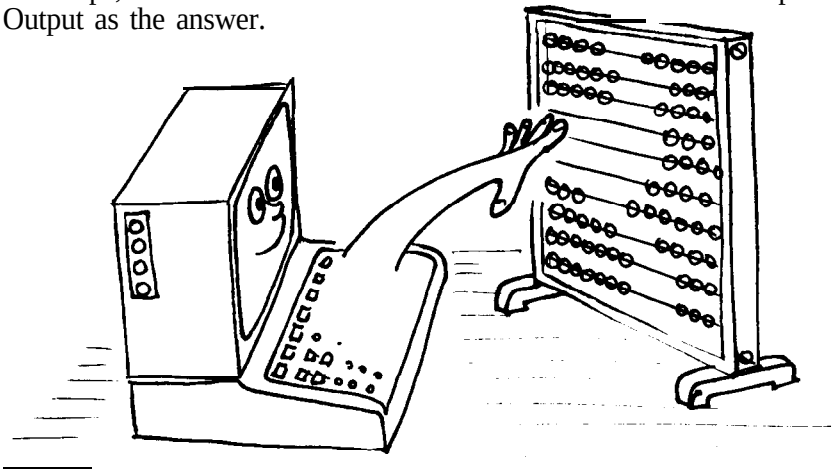
If a computer is used to store the complete works of Shakespeare, there is not a single letter or word to found anywhere in the system. Every piece of information is first converted into a number that the computer can locate and then manipulate as required.

BASIC interprets the words as numbers which the computer can then manipulate using only addition, subtraction and features from Boolean logic that permits the computer to compare data and select for certain attributes. - ie check to see if one number is bigger then or the same as another, or to perform a defined task if one number OR another meets certain criteria.

Through the medium of the program, the computer breaks down every task into a simple series of Yes/No operations.

The process of multiplication is performed using multiple additions - the BASIC instruction to multiply 35 by 10 (35*10) gets to the answer by adding 35 to itself ten times.

Part of the Central Processor Unit (CPU) is loaded with the numeric data for 10, and another part of the CPU is loaded with 35. Each time 35 is added to itself, the part of memory containing 10 is decreased by one until it reaches zero, when the process stops, and the accumulated result of 350 is sent to another part of the CPU for Output as the answer.



If this process sounds cumbersome, then you're quite right, as you have uncovered the first and most important truth about computing. A computer is primarily a tool for performing the simplest of repetetive tasks very quickly and with absolute precision, Thus BASIC interprets the instructions given in the form of the program, and translates them into the language that can be handled by the CPU. Only two states are understood by the logic of a computer - **'yes'** or 'no', represented in binary notation as '1' and '0'. The representation in Boolean logic is simply 'true' and 'false' - there's no such thing as a 'maybe' or 'perhaps'.

The process of switching between these two distinct states is the essence of the term digital, and is sometimes referred to as toggling. In the world of nature, most processes move gradually from one completely 'stable' state to another in a smooth progression. In other words, the transition is made by following the path of a line between the two states - in an ideal digital enviroment the switch from one state to the next is made in no time at all - but the physics of semiconductor science dictate that there will be some minor delay, referred to as propagation delays - and it is the accumulation of many of these propagation delays that provides the reason why a computer has to spend some time processing the information before the answer comes out.

In any case, the computer would have to wait a finite time for one task to have finished before it can start work on the result of that first task - so there would need to be some artificial delay imposed anyway. The digital process is black or white where the stages in the transition via shades of grey have no significance whatsoever. The smooth progression is via various shades of grey.

If the ultimate answer is either 0 or 1, then there is no possibility of it being 'nearly' correct. The fact that computers can sometimes appear to make errors when handling numeric data is due to the limitation of the size of numbers it can process requiring 'oversize' data to be squeezed down to fit the space available, or truncated, leading to rounding errors. eg 999,999,999 becomes 1,000,000,000.

In a world where the only two numbers available are 0 or 1, how do you then count beyond **1 ?**

Bits and Bytes

We just happen to be used to understanding numbers based on the decimal system where the reference point is the number 10 - ie there are ten digits available to represent quantities in range from 0 to 9 (which is used in preference to the expression 1 to 10). The system where numbers range from 0 to 1 is the binary system, and the units in which the system operates are called bits - an abbreviated form of **'BInary digiT'**. It might be less confusing if you forget all about the decimal context of the 0 and the 1, and use two totally different symbols to represent their function: fl and * can be used equally well, as long as there is consistency throughout.

The relationship between bits and decimal notation is simple to understand:

It's actually conventional to declare the maximum number of binary digits being used by adding leading zeros to make up the number to the full number of bits:

decimal 7 becomes
00111 binary
using 5 bit notation

In the binary system, the figures may be considered merely as indicators in columns to specify whether or not a given power 2 is present (1), or not (0).

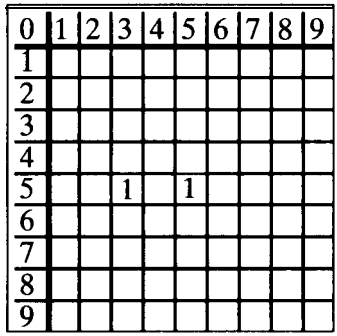
$2^0 = 1$
$2^1 = 2 = 2 = 2(2^0)$
$2^2 = 4 = 2x2 = 2(2^1)$
$2^3 = 8 = 2x2x2 = 2(2^2)$
$2^4 = 16 = 2x2x2x2 = 2(2^3)$

so the columns look like

2^4	2^3	2^2	2^1	2^0	
1	0	0	1	1	
(16	0	0	2	1)	= 19

In order to provide a shorthand method of referring to binary digit information, the term byte is used to denote 8 bits of information. The highest number that can be stored in a byte is then (binary) 11111111 - or (decimal) 255. This implies 256 actual variations, including 00000000, which is still perfectly valid data to a computer.

Computers tend to manipulate data in 8 bit multiples. 256 is not a very large number, so in order to acheieve an acceptable means of handling the memory, two bytes are used to provide a method of addressing memory which is in the form of an array, with a horizontal and vertical address by which the elements of that array can be located:



This array can locate up to (10x10) items of information using address numbers that lie in the range 1 to 9. The item stored at position 3,5 is a '1' - as is the item at 5,5.

So a binary array of 256x256 can handle 65,536 individual locations using 8 bit addresses for the vertical and horizontal axes of the array. So our '0' and '1' have progressed to being capable of identifying one of 65,536 unique elements.

The next level of shorthand for binary is the kilobyte (**kByte** or simply **'K'**) which is 1024 bytes. 1024 is the nearest binary multiple to the more familiar decimal use of the term 'kilo' - and explains why a computer described as having a '64K' memory does in fact have a memory of 65,536 bytes (64 x 1024).

Thankfully, the BASIC interpreter does all the necessary conversions for you, and it is quite possible to become a proficient programmer without a complete understanding of binary. Although an appreciation of the significance of binary will help you spot the many 'magic' or significant numbers that inevitably crop up as you work through the science of computing.

It's worth spending some effort to acquire an understanding of binary and the various significant numbers 255, 1024 etc., since it is very unlikely that these will change from the being the bedrock of computer operation in the forseeable future. The certainty and simplicity that comes from working in only two states will prevail over the enormously increased complexity that would result from any other number base.

However....

Simple and elegant as it is, binary notation is longwinded and prone to inaccuracy as it cannot be easily read at a glance. Binary has a number of associated counting systems that act as shorthand for programmers. One such number system is widely used in microcomputing called Hex (an abbreviation of hexadecimal).

Here the number is based on 16, and is represented in a single character:

Decimal

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Hex

0 1 2 3 4 5 6 7 8 9 A B C D E F

The hexadecimal system can break the eight bits of a byte into two blocks of four bits, since 15 is a four bit number: 1111 binary. The first block indicates the number of complete units of '15', and the second indicates the 'remainder' - and this is where the elegance of binary begins to emerge.

Reconsidering the table that introduced binary notation

Decimal	Binary	CPC464-ese	Hexadecimal
0	0	fl	0
1	1	*	1
2	10	*fl	2
3	11	**	3
4	100	*flfl	4
5	101	*fl*	5
6	110	**fl	6
7	111	***	7
8	1000	*flflfl	8
			9
10	1010	*flflfl	A
11	1011	*fl**	B
12	1100	**flfl	C
13	1101	**fl*	D
14	1110	***fl	E
15	1111	****	F
16	10000	*flflflfl	10

An 8-bit number 11010110 can be subdivided, and then considered as two 4-bit numbers known as nibbles , Hex D6. Throughout this guide a hex based number will be introduced by the '&' symbol eg &D6, and this is the number base most commonly used by programmers using assembly language techniques. An assembly language program is the nearest most programmers get to programming directly in machine code, since the assembly language program allows the program to use simple letter 'mnemonics' to specify the actual machine code 'numbers'.

When using HEX, you must first work out the value of the first digit to obtain the number of 16's in the final number, and then add the remainder designated by the second 'half of the hex notation to obtain the total decimal equivalent. There's a powerful temptation to regard a number like &D6 as 13+6, or 136. But it's (13x16)+(6) = 214.

It's the same process you use when you read a decimal (also known as a Denary number) number such as '89' - ie (8x10)+(9). It just happens that multiplying by ten is a great deal simpler unless you've had a lot of practise at multiplying by 16.

If you've got this far without becoming too confused, then you are well on your way to getting a grasp of the basic principles of the computer. You may even be wondering what all the fuss is about - and you'd be quite correct. A computer is a device that manages very simple concepts and ideas: it just happens to perform these tasks at great speed (millions of times per second), and with a huge capacity to remember both the data that has been input, and the intermediate results of the many thousands of very simple sums along the way to the result.

If you want to pursue the theory of your computer, there are literally thousands of books available on the subject of computing. Some will tend to leave you more confused than you were when you started reading them, and a few will actually lead you along the way by revealing the simplicity and the fundamental relationships that exist between the number systems, and the way your computer deals with them.