The Economist

Science and technology | From the archives

A layman's guide to software

In 1985, The Economist explained how computer code works on Boolean algebra

Sep 21st 1985

COMPUTERS guide spaceships to the moon, run factories and help the lovelorn to find perfect mates. How? All that computer machinery can do is add, subtract and compare numbers: it is the programs that make computers clever. The jargon of the software industry often obscures the principles that lie behind computer programs. Here is a refresher course for the baffled.

A digital computer, remember, is merely a bundle of transistors, each of which can be switched on or off by adding or removing an electric charge. On and off can be interpreted in several ways: for example, as the ones and zeroes of binary counting (for storing numbers); or as the "true" and "false" of Boolean algebra (for performing logical steps such as if X and Y then Z).

Programs tell computers how to apply the logic to the numbers. Writing a simple program in a programming language like Basic is easy, but every step must be made explicit. To find the average of a set of numbers, for example, you would add all the numbers together and divide by the number of numbers. To find out whether it has added up all the numbers, the program keeps looping back to see whether the number of numbers in its counter has reached the number of numbers it has been told to expect. Only when it does can the program proceed beyond the loop to divide the running total by the number of numbers. Loops and jumps are common features of computer programs.

Crushingly obvious? Remember that programming languages like Basic had to be invented: they did not fall from heaven. Like all software, Basic ultimately consists of a series of instructions in machine code, the most primitive form of software. Each microprocessor has its own machine code, whose statements correspond directly to the operations performed by the chips' transistors. Such operations are very basic indeed. They consist of simple orders like: "move the contents of some memory address into register AX" (registers are the areas of a microprocessor in which data is manipulated) or "add the contents of register AX to register BX".

Machine code instructions take the form of long strings of ones and zeroes. The machine code statement that, for example, tells the Intel 8088 chip to add the number 10 to one of its registers is: 00000101 00001010. To tell the chip to perform the Basic statement a=b+c, one would have to write at least three such statements (one to move "b" into a register, one to add "c" to it, and a third to move the result into “a”)—and possibly more, depending on the sorts of numbers to be added.

Machine code is a colossal bore. Writing thousands of ones and zeroes without making a mistake is hard. What is worse, a program written in machine code for a computer using one kind of processor has to be written all over again to run on a computer that uses a different processor.

There is another disadvantage. Using machine code to tell a computer exactly what to do focuses the programmer’s attention on what is happening inside the computer, not on how to solve the problem at hand. It is a bit like driving a car by thinking about the proportions of air and fuel in the carburettor or about the hydraulic system that controls the brakes, instead of about the traffic on the road ahead.

The inadequacies of machine code have stimulated the development of many kinds of programming languages. As a first step, programmers tried to escape from the tedium of machine code by replacing the unmanageable binary digits with mnemonics and symbols to specify operations and memory addresses. Programs written this way are called assembly languages. At first they were translated into machine code by hand. Now their mnemonic instructions are converted into machine code by automatic programs called assemblers. While assembly languages are simply more manageable forms of machine code, programmers devised higher-level languages in order to:

• Write programs that could be run on many computers using many different kinds of processors. When IBM developed Fortran, the first high-level language, in the 1950s, it became an industry standard. Computer makers competing with IBM did not try to devise rival languages. Instead

they adopted Fortran for their own computers, by selling their machines with compilers—programs that could automatically translate Fortran instructions into code their machines could recognise.

Translation from higher-level languages to machine code can be done in two different ways: by a compiler, which translates an entire language-written program into machine code before executing it, or by an interpreter which translates and executes the programs one instruction at a time. Interpreters are generally easier to work with, but they are slow. Because the interpreter must be present when the program is run it takes up valuable space in memory. Interpreters also needlessly repeat the translation of identical looped instructions every time they are encountered.

• Get away from thinking about procedures in the machine and concentrate on problems in the world. An extreme case is the attempt by newer declarative languages to get away from instructions altogether. But all the so-called "high-level" languages—even humble Basic—push the detailed execution of tasks into the background and allow the programmer to write in shorthand.

To understand, look again at the average-finding program. Instruction number two—Let T=0—is shorthand. It tells the computer to create a memory location, the contents of which can change in the course of the program, and to remember its address. The machine-code address is actually a long binary number, but the programmer using Basic need not bother to find out what it is; calling it T will do. Finally, it tells the computer to enter a value (in this case zero) into that location.

Basic is one of only a handful of languages to have become common software currency. Programming languages differ as much as human ones. Basic, for example, is a bit like school Latin—not altogether convenient for practical purposes but easy to learn and a good introduction to programming ideas. Fortran, the grandfather of high-level languages, is a staple of programmers producing software for scientific and engineering applications. Cobol, one of the first languages to use what looks (misleadingly) like ordinary language in its instructions, is used for business applications.

Although there are several hundred programming languages, most have common elements. They all, for example, have commands to move data around in the computer's memory, and to and from the chip's registers. All perform arithmetic operations (adding, subtracting, multiplying and dividing data in the registers) and relational operations (comparing two pieces of data and flagging whether one greater than, equal to or less than the other).

Programming languages are a means to an end. They help people write programs that tell compilers to do jobs. These jobs are that the computer industry calls applications software. They do not come only in the form of off-the-shelf spreadsheets, word processors and video games. Much applications software is tailor-made, for example to organise production in a particular factory to to steer guided missiles through space.

There is also a special category of software—operating systems—which exists to organise the work of the computer itself. Well known operating systems include Unix, AT&T’s system for

16- and 32-bit computers. The operating systems of microcomputers are responsible for essential jobs like making sure that programs are executed in the right order and, crucially, organising memory (to avoid, for example, writing data over the instructions for processing it). The operating systems of big mainframes have more to do: a typical job is to ration the tasks of the computer's central processor so that it can appear to serve many different users at the same time.

Thanks to programming languages, the software industry churns out applications programs faster than would have seemed possible when programmers worked directly with machine code. Congress's office of technology assessment estimates that about 10,000 companies, from one-man operations to divisions of big corporations, are developing software for sale. There could be as many as a million computer programmers employed in industry. Yet programmers cannot keep up with demand.

Why the bottleneck? Whereas simple programs can be written in minutes, sophisticated applications software can take teams of software designers years to produce. There are up to half-a-dozen different stages in the life cycle of a big software product. One of the most time-consuming is getting an accurate software specification from the potential user. The specification is then broken into manageable segments than can be programmed by small groups.

Once these segments have been designed they must be coded, either into a higher-level language or directly into machine code. The code must then be tested and—often the most expensive stage of all—maintained. Some software systems cost more than 20 times as much to maintain as to develop. Common culprits are badly written or badly documented programs, or unexpected changes written into programs when users modify their requirements half way through the design stage.

Big efforts are under way to clear the software bottleneck. One approach is standardisation. The idea is to boost the productivity of programmers by enabling them to work in one familiar language and draw on libraries of compatible reusable code. In this way, programmers would not, say, have to write an average-finding procedure every time one was needed; they would simply summon one from a library of prewritten routines. The Pentagon, which spends between $4 billion and $8 billion a year on software, is trying to get rid of its existing Babel of computer languages by promoting the adoption of one—Ada—for as many applications as possible.

The other approach is software engineering, an attempt to turn computer power on itself to automate the production of programs. High-level languages are themselves forms of automation. Software editors and debuggers help programmers detect errors when languages are translated into machine code. Automatic checkers, called program provers, are still in their infancy.

In Japan, software factories, set up by companies with common programming requirements, have helped productivity by reusing old machine code in new applications. America's defence department, which will need to write millions of lines of new code to implement the strategic defence initiative (star wars), is considering spending $250m over 10 years on advanced software technology.