# *MANUAL OPERATION*

## RUNNING A SIMPLE JOB.

The first computer systems were just hardware. If you wanted them to do anything, you had to do all the work other than the computation itself : you had to load all the programmes you needed into memory as they were required, ensure they had all they needed in the way of input files ( cards or paper tape in appropriate readers ) and output files ( blank cards or tape in punches, paper in printers ), and set up the starting address in the computer's address register, then set it executing.

The chapter *IN THE BEGINNING* showed you what you had to do to compile and run a Fortran programme on an IBM1620 machine, with no secondary memory ( discs or magnetic tape ). There was a lot of it, and it took a long time.

If your programme didn't work, a common practice was to load it into the computer, and follow its execution in minute detail from the console. This took a long time, and was not a very efficient way of using an expensive piece of machinery, so ways of making better use of the computer were eagerly sought.

## HOW CAN IT BE MADE FASTER ?

The most obvious time waster in this system is the sequence of manual operations, particularly as they were often performed by programmers who had little incentive to make the most efficient use of time, as they had booked the computer for twice as long as they expected to need it just in case anything went wrong. So the first step to greater efficiency was to employ a *computer operator*.

The operator's job was to make sure that the computer was used as effectively as possible. The programmers were no longer permitted to touch the computer; instead, they had to hand their work as punched tape or cards to the operator, together with instructions for running it. The old pattern of hands-on debugging was replaced by the *memory dump* : the whole of the computer's memory was printed out so that the programmer could conduct a post-mortem examination.

The advent of the operator was perhaps the first step towards the development of operating systems. It is quite typical of several further steps in a number of ways :

- You have to get something new to make it work – in this case, the operator.
- It is a mixed blessing – the computer might be more efficiently used, but the programmers have to learn new ways.
- It addresses one particular source of inefficiency – in this case, the time between successive jobs, and to a smaller degree the time between successive programmes.

This particular change, being the first, also did something unique : it made the point that using a computer involved actions of quite distinct kinds, some of which were to do with what the individual programmes did when they were executed ( the programmers' concern ), and others to do with the details of running the programme on the computer ( the operators' concern ). This was more than a theoretical distinction, and the programmers had to codify what they had been doing when they ran their programmes in order to write instructions for the operators to follow. To describe the Fortran job, with a few extra twiddly bits to make it more interesting, they might write something like this :

- Load Fortran.
- Compile my programme tape labelled FORME.
- Link in the subroutines it needs.
- Copy the linked programme to magnetic tape MYTAP.
- Run the programme using data tape B.
- Return the results to ME.

Perhaps that's the first step in the evolution of *job control languages*.

Or perhaps it's the second step. Compare these instructions ( from the programmer to the operator ) with the instructions ( from the manufacturer to the programmer ) for compiling a Fortran programme on the IBM1620. The level of detail is different, but the two sets of instructions are certainly very similar in nature. The difference is that the programmer now issues the higher-level instructions instead of executing the lower-level instructions.

## WHAT NEXT ?

We can try to do the same trick again : find a source of inefficiency, and find a way round it. The next time waster is the transition from one programme to the next – which is to say, all the details which must be handled to move from one instruction to the next in the list which the programmer gives to the operator. But all such transitions are, really, quite simple routine tasks. We have a machine which can perform simple routine tasks – it's called a computer. And it doesn't take much imagination to see the programmer's list of instructions as a sort of computer programme, albeit in a rather informal language.

With a little more imagination, we could imagine the whole lot put into a package for the computer to execute without needing any intervention by the operator. If we move to a punched card system ( because they soon became far more common than paper-tape systems, and the picture is easier to draw ), we could even think of the package as something like the diagram at the beginning of the next chapter. Now all we have to do is devise a way to get the computer to do the operator's work. That leads us on to the topic of *Monitor Systems*.

---

## QUESTIONS.

Can you see an operating system in that description ? Does an operating system have to be hardware or software ?

Are there any problems in automating the operator ?

We suggested that the advent of the operator shifted the programmer's involvement in programme execution from low-level to higher-level tasks. To what extent is the operator ( or any operating system ) analogous to a compiler for a high-level language ? Are the same arguments for and against applicable in both cases ?

---